cadence®

Xtensa® LX7 Microprocessor

Data Book

For Xtensa® LX7 Processor Cores

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Issue Date: 03/2017

RG-2017.5

PD-17-2440-11-05

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

List of Figures

List of Tables

Preface

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- $ represents your shell prompt, in user-session examples.
- **`literal_input`** indicates literal command-line input.
- `variable` indicates a user parameter.
- `literal_keyword` (in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- *`... output ...`* indicates unspecified program output.
- *`[optional-`*`variable`*`]`* indicates an optional parameter.
- **`[`**`variable`**`]`** indicates a parameter within literal square-braces.
- **`{`**`variable`**`}`** indicates a parameter within literal curly-braces.
- **`(`**`variable`**`)`** indicates a parameter within literal parentheses.
- | means *OR*.
- *`(var1 | var2)`* indicates a required choice between one of multiple parameters.
- *`[var1 | var2]`* indicates an optional choice between one of multiple parameters.
- `var1` *`[, varn]*`* indicates a list of 1 or more parameters (0 or more repetitions).
- `4'b0010` is a 4-bit value specified in binary.
- `12'o7016` is a 12-bit value specified in octal.
- `10'd4839` is a 10-bit value specified in decimal.
- `32'hff2a` or `32'HFF2A` is a 32-bit value specified in hexadecimal.

Terms

- *0x* at the beginning of a value indicates a hexadecimal value.
- *b* means bit.
- *B* means byte.
- *Mb* means megabit.
- *MB* means megabyte.
- *PC* means program counter.
- *word* means 4 bytes.

Changes from the Previous Version

The following changes were made to this document for the Cadence RG-2017.5 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Added description for HiFi 3z DSP, HiFi4 DSP, and Fusion G6 DSP in Section 9.4 “DSP Options”.
- Updated terminology usage:
  - Conditional Store Synchronization Instruction Option to Conditional Store Option in Section 4.6
  - Exclusive Access Synchronization Option to Exclusive Access Option in Section 4.7
- Added description for `DTagCheckWrData` error check bits in Table 10–20
- Clarified how the iDMA control word QoS bits get mapped on the AXI interface, see Appendix 19–71
- Added description on how AXI transmission faults affect AXI response signaling, see Section 28.6.17 and Section 28.10.11

Note: The RG-2016.4 release is the first general release of Xtensa LX7 architecture and software. Xtensa software and tools are available to all users for software upgrade from previous releases.

The following changes were made to this document for the Cadence RG-2016.4 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Updated the TMode row of Table 10–49 to clarify the configuration option required for the presence of this primary input.
- Added clarification that the conditional store synchronization option and the exclusive access synchronization option are mutually exclusive in Section 4.7.
- Added the DebugMode signal, which is the same as the XOCDMode signal but not maskable by software. See Table 10–49 and Table 35–141.
- Added a footnote to clarify that Xtensa LX7 requires the AXI bus option to be selected with iDMA, see Section 19.1.7.
- Added a bullet in Section 18.1 describing data local memory rules for power savings in configurations with the MPU.

- Added description in Section 3.3 “The Xtensa Memory Protection Unit (MPU)” clarifying internal vs. external memory types, and also the limited number of cache policies at the internal L1 cache level.
- Merged the Introduction chapter into Chapter 1 and revised the content.
- Revised text throughout various chapters for accuracy.
- Added clarification for switching the MMU/MPU attribute of the currently running code. A MEMW instruction is needed before the WITLB/WPTLB instruction. See Section 3.1, Section 3.2.2, and Section 3.3.1.
- Added bit field details for the ECC/Parity Signals for AXI4 Master Interface, AXI4 Slave Interface, and iDMA AXI4 Master Interface, see Table 10–40, Table 10–42, and Table 10–44.
- Added more details for the CACHEADRDIS register bit settings, MPU entry settings, and exception occurrence, see Section 3.3.5 and Section 3.3.6.
- Updated description for processor wait mode and iDMA activity, see Section 19.1.14

Note: In the RG-2016.3 release, Xtensa LX7 architecture and hardware are provided for a limited set of products. Xtensa software and tools are available to all users for software upgrade from previous releases.

The following changes were made to this document for the Cadence RG-2016.3 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Updated various sections in the chapter Introducing the Xtensa Processor Generator and in Chapter 1 “Introducing the Xtensa Processor” .
- Added the term and definition for bypass bufferable in Table 3–1 and amended the access mode value information in Table 3–2.
- Updated information about the S32C1I instruction in Table 4–9 and Section 4.6.2.
- Added a restriction in Section 4.7.5 “Monitors for Exclusive Access ”.
- Noted that a write of each PIF width of the prefetch RAM width is controlled separately for the Prefetch RAM PIF Width Enable signal in Table 10–26.
- Clarification about the `IHU` and `IIU` instructions unlocking cache lines in Section 16.3.
- Clarified information about the instExc bit of the MESR register in Section 16.4 “Instruction-Cache Memory Integrity” and Section 18.4 “Instruction RAM Memory Integrity”.
- Clarified information about configuration of block prefetches in Section 17.2.1.
- Added information explaining when a prefetch entry is available for allocation in Section 17.2.6 “Software Prefetch”.

- Added information about If a memory error is detected on a load from IRAM or DRAM coming from inbound DMA in Section 18.4 “Instruction RAM Memory Integrity” and Section 18.11 “Data RAM Memory Integrity”.
- Clarified information about data RAMs that are configured with either ECC or parity error checking in Section 18.9.2 “Data RAM Busy”.
- Removed redundant information about using Runstall in Section 23.3.2.
- Amended information in Table 28–114 for the description of ARPROT and AWPROT bits.
- Added Section 28.9.3 “AXI Parity / ECC Protection”, which includes the equation for generating ECC codes.
- Removed duplicated text (Chapter 31 in previous release).
- Amended a table numbering problem, which began in Section 35.2 “Processor Interface (PIF) AC Timing”.
- Updated Appendix A.8 to include information about interrupt latency causes with TLB refill operations.

Note: In the RG-2015.2 release, Xtensa LX7 architecture and hardware are provided for a limited set of products. Xtensa software and tools are available to all users for software upgrade from previous releases.

The following changes were made to this document for the Cadence RG-2015.2 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Removed redundant information about memory-management options in Chapter 23, which is covered in Chapter 3.
- Amended Table 28–114.
- Added clarification about JRST requirements in Section 35.4.4 “JTAG AC Timing”.

The following changes were made to this document for the Cadence RG-2015.1 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Added information about the Vision P5 DSP in the following sections:
  - Section “The gather/scatter option (when configured), introduces the gather/scatter interrupt. When the SuperGather engine detects an error, it causes assertion of the gather/scatter interrupt. This signal is automatically connected to the Xtensa processor and observes the same rules as the other interrupts, such as maskable by INTENABLE.”
  - Section 9.3 “Optional Function Units”
- Added Section 8.4 “ISA Implementation”

- Added information about the SuperGather option as follows:
  - Added information in Section 9.2 “Core Instruction Options”
  - Added gather/scatter register reset values in Table 9–15.
  - Added information about SuperGather sub-banks in Table 10–22 and Section 18.7 “SuperGather Sub-banks”.
  - Added information about gather/scatter interrupts in Section 2.1.10 “Gather/Scatter Interrupt” and in Section 9.5 “Exception and Interrupt Options”.
  - Information about gather/scatter hold R stalls added in Appendix A.1.4 and in Appendix A.1.6.
  - Gather/scatter global stall information added in Table A–166
- Clarified information about additional FLIX instruction widths in Section 9.2 “Core Instruction Options”.
- Added a list of features that require an MPU in Section 9.20 “Incompatible Configuration Options”.
- Amended Note #9 in Table 10–20, added Table 10–47, Coprocessor Exceptions and updated Table 10–51.
- Added Section 13.3.11 “Maintaining Outbound Requests”.
- Section 13.3.12 “Bus Errors” contains clarifications about PIF errors.
- Clarified that there is a cache line’s worth of buffering for each prefetch entry in Section 17.4.1 “Prefetch Stream Detection and Prefetch Entry Allocation/De-allocation”.
- Updated the arbitration scheme for data RAM accesses in Section 18.5 “Different Connection Options for Local Memories”.
- Clarified information about iDMA using an AXI bridge in Section 19.1.7 “AXI Transactions” and in Section 28.1 “Separate AXI Master Port for iDMA”.
- Updated load/store exception information in Section 18.9.4 “Load/Store Transaction in FLIX Packets” and Section 23.4 “Memory Transaction Ordering”.
- Amended information about asynchronous bus bridge clocks in Section 22.2 “Clocks”.
- Revised Table 32–121, Energy Reduction Through ISA Extensions.
- Updated Table 35–142.

The following changes were made to this document for the Cadence RG-2015.0 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Added a description of the Memory Protection Unit (MPU) in Section 3.3 “The Xtensa Memory Protection Unit (MPU)”.

- Added Section 4.7 “Exclusive Access Option”
- Updated Chapter 10 “Xtensa LX7 Bus and Signal Descriptions” .
- Modified Chapter 11 “Processor Interface (PIF) Main Bus” to remove AHB-Lite and AXI Bus Bridges section, which is moved to later sections.
- Added Section 17.4.3 “Prefetch Buffer Memory Errors”.
- Added a warning about using external agents in Section 18.9.2 “Data RAM Busy”.
- Added new Chapter 19 “Integrated DMA (iDMA)” .
- Amended and updated Section 22.3 “Reset”.
- Clarified information about the `ErrorCorrected` output signal in Section 22.8 “Error Corrected Signal”.
- Updated EDA tool support information in Chapter 25 “Xtensa EDA Development Environment and Hardware Implementation” .
- Added and updated information for the following chapters about the Cadence-provided bus bridges. These new chapters contain information from the former *Xtensa Processor Bus Bridges Guide.*
  - Chapter 26 “Bus Bridges”
  - Chapter 27 “AHB-Lite Bus Bridge”
  - Chapter 28 “AXI and ACE-Lite Bus Bridges” (new features, including Exclusive Stores, ACE-Lite, AXI security, and Bus ECC, are described in Chapter 28.2 “Configuration Options”)
  - Chapter 29 “Fault Handling and Error Reporting”
- Updated the test and OCD signal specifications in Table 35–141.
- Added Appendix C, External Registers.

1. Introducing the Xtensa Processor

The Tensilica IP division of Cadence Design Inc. provides SoC (system-on-chip) designers with a configurable and extensible processor fully supported by automatic hardware and software generation. The Xtensa processor is the first processor designed specifically to meet the wide range of performance requirements in today's SoC designs, covering applications requiring conventional control and DSP functionality to something more proprietary. SoC designers can add optimized processors with flexibility and longevity through software programmability, as well as differentiation through implementations tailored for the specific application.

1.1 Configurable and Extensible Processors

Although general-purpose embedded processors can handle many tasks, they often lack the bandwidth needed to perform complex, high-bandwidth data-processing tasks such as multi-channel audio, computer vision, and high speed wireless communications. Chip designers have long turned to hardwired logic (blocks of RTL) to implement these key functions. As the complexity and bandwidth requirements of electronic systems increase, the total amount of on-chip logic rises steadily.

Even as designers wrestle with the growing resource demands of advanced chip design, they face two additional worries:

1. How do design teams ensure that the SoC specification satisfies customer needs?
2. How do design teams ensure that the SoC's design meets those specifications?

Further, a good design team will anticipate future needs of current customers and potential future customers—and add a built-in road map.

If the design team fails on the first criterion, the SoC may work perfectly, but it may have inadequate sales to justify the design and manufacturing effort. Changes in requirements may be driven by demands of specific key customers, or they may reflect overall market trends such as the emergence of new data formats, new protocol standards, or new feature expectations across an entire product category. While most SoC designs include some form of embedded control processor, the limited performance of these fixed-ISA (instruction-set architecture) processors often precludes them from being used for essential data processing tasks. Consequently, RTL blocks are designed to handle these high-performance tasks and, therefore, software usually cannot be used to add or change fundamental new features after the SoC is designed and fabricated.

If the design team fails on the second criterion, additional time and resources must go towards changing or fixing the design. This resource drain delays market entry and causes companies to miss key customer commitments. Failure is most often realized as a program delay. This delay may come in the form of missed integration or verification milestones, or it may come in the form of hardware bugs—explicit logic errors that are not caught by the limited verification that is typical of hardware simulation. The underlying cause of failure might be a subtle error in a single design element or it might be a miscommunication of requirements—subtle differences in assumptions between hardware and software teams, between design and verification teams, or between the SoC designers and the SoC library or foundry supplier. In any case, the design team may often be forced to jump into an urgent cycle of re-design, re-verification, and re-fabrication (re-spinning) of the chip. These design "spins" rarely take less than less than six months, which significantly disrupts product and business plans.

1.2 Using Tensilica Processors in SoC Designs

System developers are trying to solve two closely related problems. One problem is the need to significantly reduce the resource levels required to develop systems by making it easier to design the chips in those systems. The second problem is the need to make SoCs sufficiently flexible so that every new system design does not require a new SoC design.

The way to solve these two problems is to make the SoC sufficiently flexible so that one chip design will efficiently serve 10, 100, or 1,000 different system designs, while giving up none or perhaps just a few of the benefits of SoC integration. Solving these problems means creating chips that can satisfy the requirements of current system designs and next-generation designs to amortize SoC-development costs over a larger number of systems.

The specialized nature of individual embedded applications creates two issues for general-purpose fixed function processors and DSPs in data-intensive embedded applications. First, there is a poor match between the critical functions needed by many embedded applications (for example, image, audio, and protocol processing) and a fixed-ISA processor's basic integer instruction set and register file. As a result of this mismatch, these critical embedded applications often require an unacceptable number of computation cycles when run on general-purpose processors. In other words, general-purpose processors are too slow for many of these critical tasks. Second, narrowly focused, low-cost embedded devices (such as music players, printers, digital cameras, or electronic games) cannot take full advantage of a general-purpose processor's broad capabilities. Consequently, expensive silicon resources built into the processor are wasted in these applications because they are not needed by the specific embedded tasks assigned to the processor.

Many embedded systems interact closely with the real world or communicate complex data (like neural networks) at high rates. These data-intensive tasks could be performed by some hypothetical general-purpose microprocessor running at sky-high clock rates. For many embedded tasks, however, there's no processor fast enough to serve as a practical alternative.

Even if there are general purpose processors with sufficiently high clock rates, the fastest available processors are typically too expensive and dissipate too much power (by orders of magnitude) to meet embedded-system design goals. Consequently, designers have traditionally turned to hard-wired circuits to perform these data-intensive functions.

Wide availability of logic synthesis and ASIC design tools has made RTL design the standard for hardware developers. RTL-based design is reasonably efficient (compared to custom, transistor-level circuit design) and can effectively exploit the intrinsic parallelism of many data-intensive problems. RTL design methods can often achieve tens or hundreds of times the performance achieved by a general-purpose processor running application firmware, often at lower power.

Like RTL-based designs using logic synthesis, optimizable processors enable the design of high-speed logic blocks tailored to the assigned task. The key difference is that RTL state machines are implemented in hardware, whereas logic blocks based on optimizable processors implement their state machines with firmware. Firmware is better at dealing with complex algorithms, faster to simulate, much easier to debug, and can be changed if market requirements or protocol standards change.

1.3 Benefitting from Processor Optimization

A fully featured configurable and extensible processor consists of a base, or foundation, processor design and a design-tool environment that permits significant adaptation of that base processor. By allowing system designers to change major processor functions, the processor can be tuned for specific applications. Typical forms of configurability include additions, deletions, and modifications to memories, to external-bus widths and handshake protocols, and to commonly used processor peripherals. An important superset of configurable processors is the extensible processor—a processor whose functions, especially its instruction set, can be extended by the application developer to include features never considered or imagined by designers of the original processor.

Changing the processor's instruction set, memories, and interfaces can make a significant difference in its efficiency and performance, particularly for the data-intensive applications that represent the "heavy-lifting" of many embedded systems. These features might be too narrowly used to justify inclusion in a general-purpose instruction set and hand-designed processor hardware. All general-purpose processors are compromised designs, where features that provide modest benefits to all customers supersede features that provide dramatic benefits to a few. Design compromises are necessary be-

cause the historic costs and the difficulty of manual processor design mandate that only a few different processor designs can be built. Automatic processor generation dramatically reduces processor-development cost and development time so that inclusion of designer-defined application-specific features to meet performance goals and deletion of unused features to meet cost goals suddenly becomes a very attractive alternative for system designers.

A configurable processor is a processor whose features can be "pruned" or augmented by parametric selection. Configurable processors may be implemented in many different hardware forms, ranging from ASICs with hardware implementation times of many weeks to FPGAs with implementation times of minutes. Extensible processors constitute an important superset of configurable processors where they can be extended by the application developer to include instructions, execution units, registers, I/O, and other features that differentiate their design from all others.

The usefulness of both configurable and extensible processors is strongly tied to the automatic availability of both the hardware implementation and the development environment supporting all configurability and extensibility aspects. Automated software support for extended features is especially important. Configuration or extension of the hardware without complementary enhancement of the compiler, assembler, simulator, debugger, real-time operating systems, and other software-development tools essentially leave the promises of performance and flexibility unfulfilled because the newly created processor cannot be programmed.

Extensibility's goal is to allow features to be added or adapted in any form that optimizes the processor’s cost, power consumption, and application performance.

The emergence of configurable and extensible application-specific processors creates a new design path that is quick and easy enough for development and refinement of new protocols and standards, fast enough to meet the application’s performance demands, and efficient enough in silicon area and power consumption to permit very high volume deployment.

1.4 Introducing the Xtensa Processor Generator

The Xtensa synthesizable processor was the first and remains the leading configurable and extensible processor architecture designed specifically to address embedded System-On-Chip (SoC) applications. The Xtensa architecture was designed from the start to be configurable, which allows designers to precisely tailor each processor implementation—often more than one different unique implementation in each chip—to match the target SoC’s application requirements. Members of the Xtensa processor family are unlike conventional embedded processors—they change the rules of the SoC game. Using Xtensa technology, the system designer molds each processor to fit its assigned tasks on the SoC by selecting and configuring predefined architectural elements and by in-

venting completely new instructions and hardware execution units that can deliver application-specific performance levels that are orders of magnitude faster than alternative processor or DSP solutions.

Cadence empowers the SoC designer to mix and match the best attributes of (1) conventional microprocessors (CPUs); (2) conventional digital signal processors (DSPs); combined with (3) the application-specific hardware acceleration familiar to hardware RTL developers.

Designers perform their optimization and create their ideal Tensilica processor by using the Xtensa Processor Generator. In addition to producing the processor hardware RTL, the Xtensa Processor Generator automatically generates a complete, optimized software-development environment that includes a full tool chain, sophisticated C and System C models, and even customized operating system support for each processor configuration. The power and flexibility of the configurable Xtensa processor family make it the ideal choice for complex SoC designs.

The Xtensa processor architecture consists of hundreds of standard, optional and configurable building blocks. Common to all configurations is the Xtensa base instruction set architecture (ISA). Configurable function blocks are elements parameterized by the system designer. Optional function blocks indicate elements available to accelerate specific applications. Many optional and configurable blocks have user-configurable sub-elements (such as timer interrupts and interrupt levels) that can be individually scaled to fit specific applications. SoC hardware and firmware designers can add advanced functions to the processor's architecture in the form of hardware execution units and registers to accelerate specific algorithms.

Cadence delivers five technologies to help designers build SoCs for embedded applications:

1. The Xtensa processor architecture based on a highly configurable, extensible, and synthesizable 32-bit processor. Many designer-defined, application-specific families of processors can be built around the base Xtensa ISA to optimize factors such as code size, die size, application performance, and power dissipation. Designers define new processor instructions, execution units, registers, and I/O using the Tensilica Instruction Extension (TIE) language.
2. A generated software tool suite to match the configured processor architecture. This tool suite includes the Xtensa C/C++ compiler (XCC), a macro assembler, linker, debugger, diagnostics, reference test benches, and a basic software library. XCC provides C++ capabilities equivalent to the GNU C++ compiler version 4.2. It improves code performance relative to GCC in many cases and provides vectorization support for our families of audio, communications and imaging/vision DSPs.
3. Xtensa Xplorer integrated design environment (IDE), which serves as a cockpit for single- and multiple-processor SoC hardware and software design. Xtensa Xplorer integrates software development, processor optimization and multiple-processor

SoC architecture tools into one common design environment. It also integrates SoC simulation and analysis tools. Xtensa Xplorer is a visual environment with a host of automation tools that makes creating Xtensa processor-based SoC hardware and software much easier. Xplorer supports basic design management, invocation of processor configuration tools (the Xtensa Processor Generator and the TIE compiler), and software development tools. Xtensa Xplorer is particularly useful for the development of TIE instructions that maximize performance for a particular application. Different Xtensa processor configurations and TIE extensions can be saved, profiled against the target C/C++ software, and compared.
Xtensa Xplorer includes automated graphing tools that create spreadsheet-style comparison charts of performance.

4. A multiple processor (MP)-capable instruction set simulator (ISS) and C/C++ callable simulation libraries.
5. A wide portfolio of application optimized and multi-purpose DSP options that can be further enhanced via the TIE language.

All development tools are automatically built to match the exact configuration specified in the Xtensa Processor Generator. Together, these technologies establish an improved method for rapid design, verification, and integration of application-specific hardware and software.

TIE language is used to describe designer-defined instructions, new registers, execution units, and I/O that are then automatically added to the Xtensa processor by the Xtensa Processor Generator. Aggressive use of parallelism and other techniques in TIE extensions can often deliver 10X, 100X or even greater performance increases compared to conventional fixed instruction set processors or DSPs.

1.5 Ease of Configuration with the Xtensa Processor Generator

The Xtensa Processor Generator assists designers in creating application-specific embedded processors rapidly and reliably. By combining the Xtensa processor architecture with electronic design automation (EDA) and embedded software development technology, the Xtensa Processor Generator allows developers to create optimized designs with unique characteristics for their application-specific processor architecture. Even developers with no processor design experience can use the Xtensa Processor Generator to develop highly tuned, application-specific processors for SoC designs.

The generator allows for easy selection and implementation of many processor features to match the target application. For example, a designer can define processor attributes such as exception configurations, interrupt-level settings, memory hierarchy, processor interface size, write-buffer size, on-chip RAM and ROM size, cache configurations,

cache attributes, register-set size, and optional DSP functions. Additionally, designer-defined instructions can be incorporated quickly to maximize application performance and reduce code size.

1.6 Ease of Integration

To facilitate SoC development, the Xtensa Processor Generator automatically generates the following outputs: the RTL code base, static-timing model, hardware-design interface, configured software tool chain, profiler, initialization and self-test code, customized instruction-set simulator, customized pin-level simulator for high speed co-simulation using standard RTL simulators, and third-party RTOS support. Cadence supports the major EDA vendor tools used in industry-standard semiconductor design flows. For more information on currently supported EDA tools and flows, see the *Xtensa Hardware User's Guide*.

1.7 Xtensa Processor Based DSPs

When configuring the Xtensa processor, you may select one of several DSP options. For information on the Xtensa processor based DSPs, see the specific reference document.

For details about the audio family of DSPs, HiFi DSPs, see the following:

- *HiFi Mini Audio Engine User's Guide*
- *HiFi 2/EP Audio Engine User's Guide*
- *HiFi 3 Audio Engine User's Guide*
- *HiFi 4 Audio Engine User's Guide*

For details about the communications family of DSPs, ConnX DSPs, see the following guides:

- *ConnX BBE16EP User's Guide*
- *ConnX BBE32EP User's Guide*
- *ConnX BBE64EP User's Guide*

For details about the Fusion family of DSPs, see the following:

- *Fusion F1 DSP User's Guide*
- *Fusion G3 DSP User's Guide*

For details about the imaging/vision family of DSPs, Vision DSPs, see the following:

- *Vision P5 DSP User's Guide*
- *Vision P6 DSP User's Guide*

2. Exception-Processing States and Options

The processor implements basic functions needed to manage both *synchronous exceptions* and *asynchronous exceptions* (interrupts). Synchronous exceptions are those caused by illegal instructions, system calls, instruction-fetch errors, and load/store errors. Asynchronous exceptions are generated through the interrupt, high-priority interrupt, and timer signals.

The processor can implement 32 interrupts. Each interrupt is configured with a level and a type. Types include edge or level triggered, software, timer, non-maskable and error reporting. Level-1 interrupts are asynchronous exceptions triggered by processor input signals or software exceptions. Level-1 interrupts have the lowest priority of all interrupts and are handled differently than high-priority interrupts. The interrupt option is a prerequisite for the high-priority-interrupt, peripheral-timer, and debug options.

The high-priority-interrupt option implements a configurable number of interrupt levels, 2 through 6). In addition, an optional non-maskable interrupt (NMI) has an implicit infinite priority level. Like Level-1 interrupts, high-priority interrupts can be external or software exceptions. Unlike Level-1 interrupts, each high-priority interrupt level has its own interrupt vector and dedicated registers for saving processor state. These separate interrupt vectors and special registers permit the creation of very efficient handler mechanisms that may need only a few lines of assembler code, avoiding the need for a subroutine call.

2.1 Exception Architecture

The processor supports the latest exception model, Xtensa Exception Architecture 2 (XEA2). The exception-handling process saves the address of the instruction causing the exception into special register EPC[1] and the cause code for the exception into special register EXCCAUSE. Interrupts and exceptions are precise, so that on returning from the exception handler, program execution can continue exactly where it left off. (Note: Of necessity, write bus-error exceptions are not precise.)

2.1.1 Exception Vectors and the Relocatable Vectors Option

The position of exception vectors is determined when an Xtensa processor is configured and generated, however the Relocatable Vectors option offers some run-time flexibility on the exact location of the vectors.

Without the Relocatable Vectors option, all vectors are statically determined when a processor is configured and generated. This model is generally adequate for most applications, and is the default.

When the Relocatable Vectors option is selected, some run-time flexibility exists in the position of vectors. The vectors are divided into two groups. The first group, the Static Vector Group, includes the reset and memory-error handler vectors. The second group, the Relocatable Vectors Group, includes all other exception vectors, including the user, kernel, double, window overflow, window underflow, and high-priority interrupt vectors.

The Static Vector group offers a choice between two vector locations that is selected by an input port, `StatVectorSel`. By default, these two locations are determined when a processor is configured and generated. There is also a sub-option under this Relocatable Vectors option, called the External Reset Vector option. This sub-option adds an input bus to the processor interface pins, which allows the user to provide an alternate reset vector. For more information, see Section 22.4 “Static Vector Select” on page 416.

The Relocatable Vectors Group’s vectors are configured with fixed relative offsets from a vector base address. The vector base address is derived from the VECBASE special register, and can be modified by software.

2.1.2 Unaligned Access Options

Unaligned Exceptions allow an exception to be raised upon any unaligned memory access whether it is generated by core processor memory instructions, by optional instructions, or by designer-defined TIE instructions.

Note that instructions that deal with the cache lines, such as prefetch and cache-management, will not cause unaligned exceptions. The Unaligned Handled By Hardware option performs unaligned accesses through multiple memory accesses to sequential locations under the control of a hardware state machine in the processor. (Some unusual cases of unaligned access will still result in taking an exception.)

2.1.3 Level-1 Interrupts

External interrupt inputs can be level-sensitive or edge-triggered. The processor can test the state of any interrupt inputs at any time by reading the `INTERRUPT` register. An arbitrary number of software interrupts (up to a total of 32 for all types of interrupts) that are not tied to an external signal can also be configured. These interrupts use the general exception/interrupt handler. Software can then manipulate the bits of interrupt enable fields to prioritize the exceptions. The processor accepts an interrupt when an `interrupt` signal is asserted and the proper interrupt enable bits are set in the `INTENABLE` register and in the `INTLEVEL` field of the PS register.

2.1.4 High-Priority Interrupts

The processor can have up to six level-sensitive or edge-triggered high-priority interrupt levels. One NMI or non-maskable interrupt can also be configured. Each high-priority interrupt level has its own interrupt vector and its own dedicated set of three special registers (EPC, EPS, and EXCSAVE) used to save the processor state.

2.1.5 C-Callable Interrupt Handlers

The processor's interrupt structure supports efficient interrupt handling in C. Note that for High-Priority interrupt sources, efficient handling in C requires the use of the call0 ABI. For more information about interrupt dispatch, see the *Xtensa Microprocessor Programmer's Guide*.

2.1.6 Timer Interrupts

The processor can have up to three hardware timers that can be used to generate periodic interrupts. Timers can be configured to generate either a level-1 or high-level interrupts. Software manages timers through one special register that contains the processor's current clock count (which increments each clock cycle) and additional special registers (one for each timer) for setting match-count values that determine when the next timer interrupt will occur.

2.1.7 Local-Memory Parity and ECC Options

The processor's local instruction RAMs, local data RAMs, prefetch buffers, and memory caches can optionally be configured with memory parity or error-correcting code (ECC). These options add extra memory bits that are read from and written to the RAMs and caches. The processor uses the extra bits to detect errors that may occur in the cache data (and sometimes, to correct these errors as well). The parity option allows the processor to detect single-bit errors. The ECC option allows the processor to detect and correct single-bit errors and to detect double-bit errors.

The processor will take a memory-error exception under the following conditions:

- If a memory error is detected on an instruction fetch, either correctable or uncorrectable, an exception is signalled when an instruction decoding uses a byte from the fetched word. See Appendix A "Xtensa Pipeline and Memory Performance" for more details.
- If an uncorrectable memory error is detected on a data load, an exception is signalled when the load instruction uses a byte from the word read.

- If a correctable memory error is detected on a data load, an exception may be signalled when the load instruction uses a byte from the word read. The Data Exception field in the MESR register controls if the exception is generated. The exception occurs in the W-stage of the affected instruction just like any other exception.
- On a load from instruction RAM, if an uncorrectable error is detected or if a correctable error occurs but the `DataExc` bit of the `MESR` register is set: a memory error exception will be signaled in the W-stage on the replay (refer to Appendix A.4 for replay explanation) of the instruction.

2.1.8 Profiling Interrupt

The Performance Monitor option (when configured), introduces the Profiling Type of interrupt. Counter overflow causes assertion of an interrupt signal to the Xtensa processor. The signal is automatically connected to the Xtensa processor, and becomes one of the 32 allowed interrupts as described in Section 2.1.3 “Level-1 Interrupts”, and observes the same rules as the other interrupts, such as maskable by INTENABLE.

2.1.9 iDMA Interrupt

The iDMA option (when configured), introduces the iDMAError and iDMADone interrupts. When iDMA runs into an error, it causes assertion of the iDMAError interrupt. An iDMA descriptor can be programmed to trigger a iDMADone interrupt upon the completion of that descriptor. These signals are automatically connected to the Xtensa processor and observes the same rules as the other interrupts, such as maskable by INTENABLE.

2.1.10 Gather/Scatter Interrupt

The gather/scatter option (when configured), introduces the gather/scatter interrupt. When the SuperGather engine detects an error, it causes assertion of the gather/scatter interrupt. This signal is automatically connected to the Xtensa processor and observes the same rules as the other interrupts, such as maskable by INTENABLE.

3. Memory-Management Model Types

The Xtensa processor core offers the following memory-management choices:

- Region Protection:
  This option provides an instruction TLB (ITLB) and a data TLB (DTLB) that partition the processor's address space into eight memory regions of 512 MB each. Access modes of each memory region are controlled by the appropriate TLB entries. Virtual and physical addresses are identical; there is no translation between them.
- Region Protection with Translation:
  This configuration option is similar to the region-protection option described above but it also provides virtual-to-physical address translation for each of the eight memory regions.
- MMU with Translation Look Aside Buffer (TLB) and Autorefill:
  This is a general-purpose MMU that provides memory-management resources required by certain operating systems such as Linux. The MMU provides for demand paging and memory protection.
- Memory Protection Unit (MPU).
  This option provides several choices for memory protection and memory types, but no translation. The number of regions is configurable, as are the starting address and size of each region. This option is well suited for designs that require something other than eight regions and require more flexibility, but which do not need the full feature set of a Linux-compatible MMU. See Section 3.3 for full details about this option.

Table 3–1 defines memory management terms used in this document.

Table 3–1. Memory Management Terminology

Terms	Definition
Bypass	The processor will never use the cache contents for the fetches or loads. The cache is never updated on a store, regardless of whether the operation hits or misses in the cache.
Bypass bufferable	A bufferable request can receive a response from an intermediate point in the response network, a non-bufferable must receive the response from the final destination.
Bypass Interruptible	This is a read bypass request that is interruptible. An interrupt that arrives while the Xtensa processor is waiting for a read response will abort the read request. Since read requests may have long latencies, an interruptible read request can result in lower interrupt latencies. However, an interruptible read request will typically be re-issued upon return from an interrupt. This may cause problems with read-side-effect devices. Note that writes are never interruptible. On the other hand, performance is seldom impacted by write latency.

Table 3–1. Memory Management Terminology

Terms	Definition
Bypass Non-Interruptible	This is a read bypass request that is not interruptible. An interrupt arriving while the Xtensa processor is waiting for read results will not abort the read request. Hence, the servicing of the interrupt must wait until the read response arrives. Note that writes are never interruptible. On the other hand, performance is seldom impacted by write latency.
No Allocate	If an access hits in the cache, the processor will use or update the contents of the cache. If an access misses in the cache, the processor will not allocate a line in the cache. It directly uses (for a load or fetch) or updates (for a store) the contents at the target memory address.
Allocate	If an access hits in the cache, the processor will use or update the contents of the cache. If an access misses in the cache, it will first cause the processor to perform a cache fill from memory and will then use (for a load or fetch) or update (for a store) the cache contents.
Write-through/ no write allocate	If a store misses in the cache, it will only update main memory and will not allocate a line in the cache. If a store hits in the cache, it will update the cache and write to main memory as well.
Write-back / no write allocate	If a store misses in the cache, it will only update main memory and it will not allocate a line in the cache. If a store hits in the cache, it will update the cache contents; the store will not go out to main memory immediately. Main memory is updated when the associated cache line is written to memory.
Write-back / write allocate	If a store misses in the cache, it will allocate a line in the cache and update the cache contents. If a store hits in the cache, it will update the cache contents. Stores will not go out to main memory immediately. Main memory is updated when the associated cache line is written to memory.
Page Table Entry (PTE)	An entry in the PageTable that specifies the physical page number, page size, ring information, and memory access modes for a page in a memory. The MMU hardware refills the Translation Lookaside Buffer from PTE.
Isolate	The processor will always use the cache for loads, regardless of cache hit or miss. Stores will always update the cache contents, regardless of cache hit or miss.
Identity Map	A mapping where virtual and physical addresses are equivalent.
Aliasing	The caches are indexed with a virtual address and tagged with a physical address, as in the Xtensa processor. It is possible under certain cache configurations that the same piece of data (from the same physical address) could reside in multiple cache sets. These sets are potentially ‘aliased’.
Static	A static TLB entry is fixed in hardware and cannot be changed by writes to the TLB.
Wired	In a wired way (or entry) the mapping can be changed, either in part or in whole. However, it will not participate in the autorefill replacement. Therefore, once software writes the entry, it is guaranteed to exist until software changes it. These are sometimes also referred to as *locked* entries or ways.
Auto-refill	Ways of the TLBs that will be loaded by hardware from the pagetable on TLB misses.

3.1 Region Protection

As shown in Figure 3–1, the ITLB and DTLB provided by the Region Protection Unit (RPU) configuration option divides the processor's memory space into eight equally sized regions. In each region, the virtual and physical addresses are equivalent. The access modes of each memory region can be changed by issuing the appropriate WITLB or WDTLB instruction. This region protection does not implement rings or ASIDs (Address Space Identifiers). Consequently, all of the Xtensa processor instructions, including privileged instructions, are always accessible.

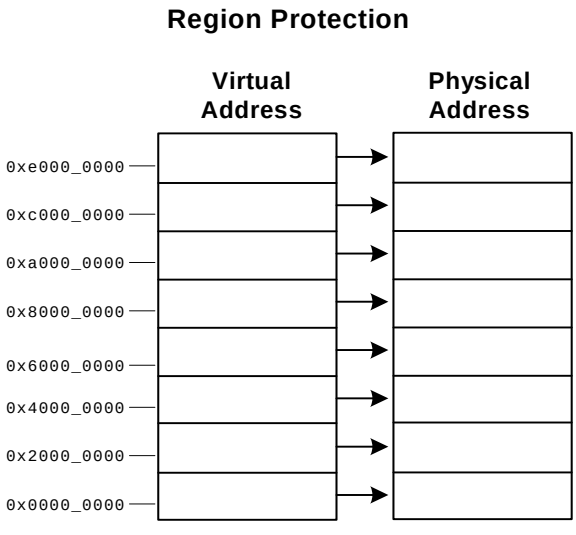


Figure 3–1. Memory Map for Region Protection Option

As shown in Figure 3–2, the Region-Protection-with-Translation configuration option is similar to the Region-Protection option, but it adds virtual-to-physical address translation for the eight memory regions. This configuration option can be used to set up two mappings to the same physical address, each with different memory-access permissions. On processor reset, all of the translations are mapped to Identity (no translation).

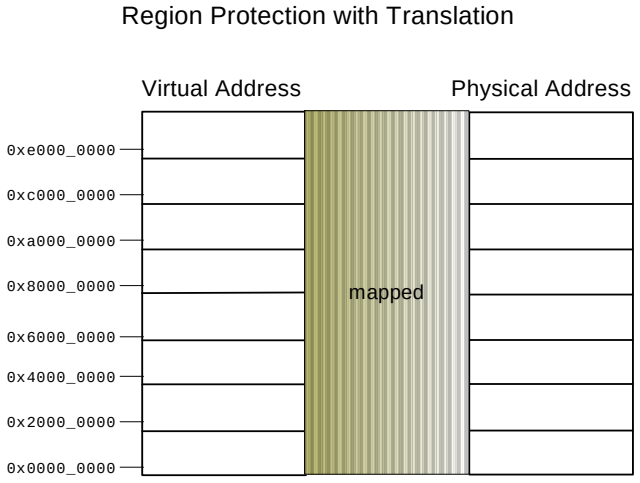


Figure 3–2. Memory Map for Region Protection with Translation Option

Note that special care is required if an instruction changes the MMU entry for the running code. The code must meet the following restriction. A MEMW instruction must precede the WITLB instruction. If running cached, MEMW and WITLB instructions must be in the same cache line.

3.1.1 TLB Configuration

Both the ITLB and the DTLB each have one way with eight entries. For the Region-Protection configuration option, VPNs (virtual page numbers) and PPNs (physical page numbers) are permanently hardwired to the identity map and the Access Mode field is writable. For the Region-Protection-with-Translation configuration option, both the PPNs and the Access Mode fields are writable, allowing for simple memory mapping.

3.1.2 Access Modes

Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for a description of the memory attributes, which are identical to the access-mode semantics. Table 3–2 lists and describes the supported access modes for instruction and data regions.

Table 3–2. Access Modes for Region-Protection (with and without Translation)

Access Mode Value	Name	Instruction Fetch Behavior	Load Behavior	Store Behavior
0000	No allocate	Exception (InstFetchProhibitedCause)	No Allocate	Write-through/No write allocate
0001	Write-through/No write allocate	Allocate	Allocate	Write-through/No write allocate
0010	Bypass	Bypass	Bypass	Bypass
0011*	Mode Not Supported	Undefined	Exception (LoadProhibitedCause)	Exception (StoreProhibitedCause)
0100	Write-back / write allocate (with Write-back cache configuration option)	Allocate	Allocate	Write-back/Write allocate
	Write-back / write allocate (without Write-back cache configuration option)	Allocate	Allocate	Write-through/No write allocate
0101	Write-back/no write allocate (with Write-back cache configuration option)	Allocate	Allocate	Write-back/No write allocate
	Write-back/no write allocate (without Write-back cache configuration option)	Allocate	Allocate	Write-through/No write allocate
0110	Bypass bufferable	Bypass	Bypass	Bypass
0111 - 1101	Reserved	Exception (InstFetchProhibitedCause)	Exception (LoadProhibitedCause)	Exception (StoreProhibitedCause)
1110	Isolate	Exception (InstFetchProhibitedCause)	Isolate	Isolate
1111	Illegal	Exception (InstFetchProhibitedCause)	Exception (LoadProhibitedCause)	Exception (StoreProhibitedCause)

Notes:

RESET VALUE: 0010 (Bypass) for all regions.

* Access Mode 3 is not supported in theXtensa LX7 core.

An exception will occur upon attempted access to a protected region when the associated access-mode attribute is invalid for the attempted operation. For additional details on exception causes, refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

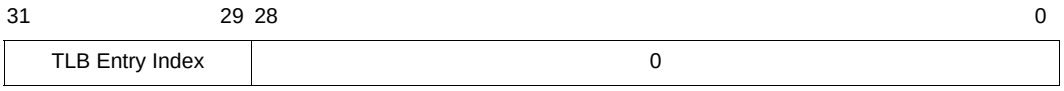
3.1.3 Protections

Each of the eight memory segments is protected through the access-mode settings written into the TLBs. Instruction fetches are not protected because the Region-Protection and Region-Protection-with-Translation configuration options do not have protection rings or ASIDs (address-space identifiers).

3.1.4 Implementation-Specific Instruction Formats

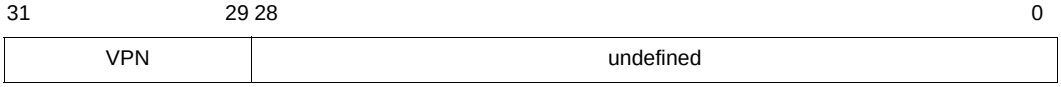
RxTLB0, RxTLB1, and WxTLB are implementation-specific instructions intended for reading and writing the contents of the instruction and data TLBs. For a complete listing of TLB instructions, refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

The format of the `as` register used for these instructions is as follows. The upper three bits are used as an index among the TLB entries just as they would be when addressing memory. They are the Virtual Page Number (VPN) or upper three bits of virtual address. The remaining bits are ignored.



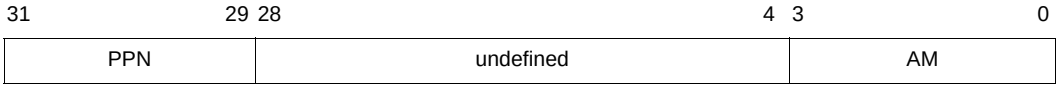
RITLB0 / RDTLB0

The RITLB0 and RDTLB0 instructions do not provide any useful information, but are mentioned here for completeness. They read the VPN out of the specified TLB entry into the `at` register in the following form:



RITLB1 / RDTLB1

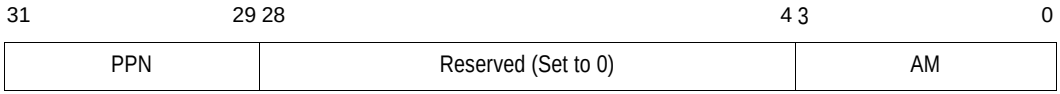
The RITLB1 and RDTLB1 instructions read the PPN and Access Modes (AM) out of the specified TLB entry into the `at` register in the following form:



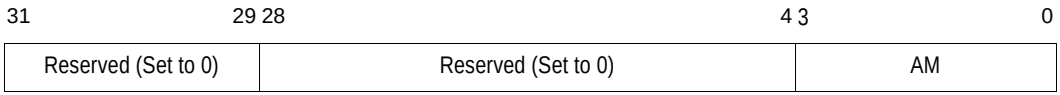
WITLB / WDTLB

The WITLB and WDTLB instructions write the contents of address register `at` to the TLB entry specified by address register `as`.

The `at` register format for region protection with translation:



The `at` register format for region protection with no translation:



The only MMU exceptions that can be taken with the Region-Protection and Region-Protection-with-Translation configuration options are: `InstFetchProhibitedCause`, `LoadProhibitedCause`, and `StoreProhibited`.

3.2 MMU with Translation Look Aside Buffer Details

The Xtensa MMU configuration option is a general-purpose Memory Management Unit that provides demand paging and memory protection. It is designed to support protected operating systems such as Linux. The MMU is configured with 4 rings (privilege levels), 256 ASIDs (address-space identifiers), and a set of asymmetric multi-way set-associative Translation Look-aside Buffers (TLBs, described in the section on “TLB Configuration” on page 21). The combination of rings and ASIDs give the Xtensa LX7 MMU four privilege levels for safely isolating multiple concurrent processes. (For a detailed explanation of the protection schema provided by ASIDs and rings, refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.)

At reset, as depicted in Figure 3–3, the Xtensa LX7 MMU maps the entire 4 GB virtual space to physical space without translation with all eight entries in way 6 of the ITLB and DTLB, set to a 512 MB page size. Each such entry is reset with PPN = VPN, ASID = 1 (the kernel ASID), and AM = 3 (bypass cache, read/write/execute enabled). All entries of other TLB ways reset to invalid (ASID = 0). This reset state is functionally equivalent to that of processors with Region Protection with Translation, except that it involves TLB

way 6 rather than a single TLB way 0. Thus, operating systems and run times not using the MMU need not be particularly aware of its presence, and can ignore the distinction between virtual and physical address spaces.

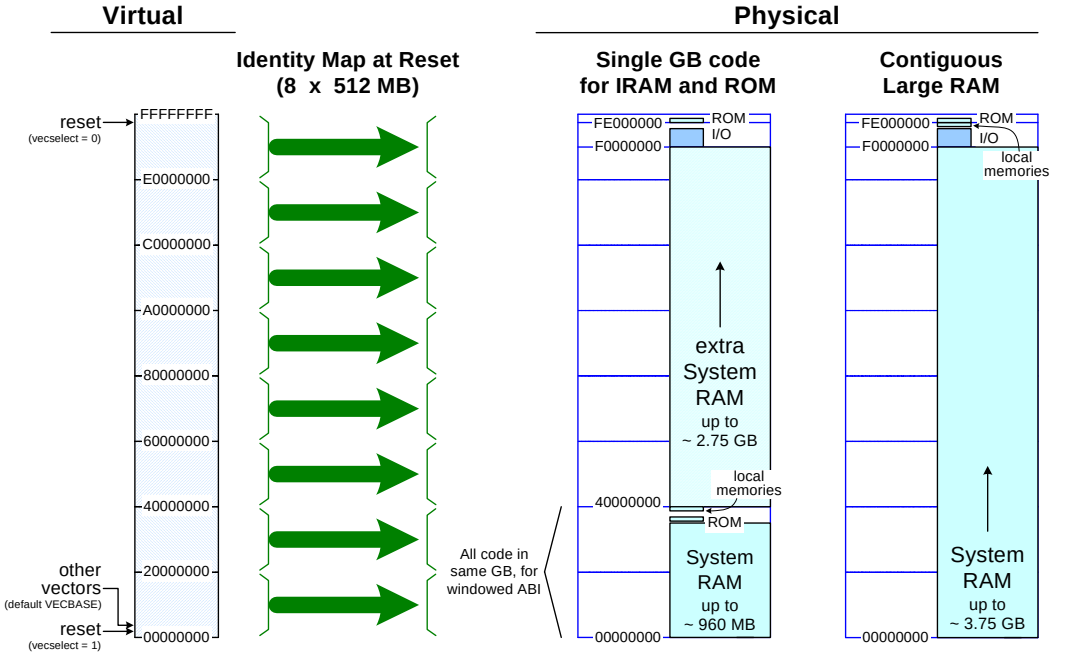


Figure 3–3. Reset Map and Physical Memory Layout Options with MMU Version 3

A processor using an MMU for running protected operating systems, such as Linux, also needs a properly laid out physical memory map. Figure 3–3 illustrates two alternative physical memory maps. Any system incorporating an Xtensa processor with an MMU must follow one of these layouts to support Linux (and similar operating systems). The default layout, labeled *Contiguous Large RAM* (on the right), allows contiguous space for system RAM to grow to nearly the full address space without gaps.

It is often desirable to satisfy the requirements of both Linux and other software such as bootloaders, RTOS, and simpler runtimes. However, the default layout has limitations for the latter software, when using the windowed ABI. The windowed ABI stores the call window size in the upper two bits of the return PC, and thus limits code to a single GB of address space (note: the call0 ABI does not have this limitation). The default layout places local memories and ROM in a different GB than the start of system RAM, where most code is likely to reside, thus preventing function calls between system RAM and either local memories or ROM. While it is possible to adjust the MMU mappings in the XPG to bring local memories and ROM down to the first GB, another alternative is provided, labeled in Figure 3–3 as *Single GB Code for IRAM and ROM*. This option allows local

memories to be placed below 0x40000000 rather than under 0xFE000000. The alternate reset vector (shown in Figure 3–3 as *reset* with vecselect=1) can also be moved, so it is possible to place a ROM in the first GB and place the alternate reset vector in it.[1]

Systems that require very large I/O maps (such as for PCI, other external busses, large flash, or other memories) that don't fit in the available 200 or so MB, may grow the I/O section downwards accordingly. In this case, grouping together devices that need cached vs. cache-bypass access, and aligning the groups to power of two sizes that match available TLB page sizes, helps ensure I/O can be mapped using wired TLB entries (simpler and faster) rather than having to use page tables.

3.2.1 TLB Configuration

Table 3–3 describes the configuration and purpose of each TLB way in the MMU-with-TLB configuration. Figure 3–4 depicts the same information visually, with details on the size of individual fields within each TLB entry.

Table 3–3. MMU with TLB Configuration

Way(s)	TLB	Purpose	Entries (indices)
0 - 3	ITLB,DTLB	4 KB pages auto-refillable from the page table. Essentially a page table entry cache.	Configurable to either 4 or 8 per way (gives either 16 or 32 refillable entries per TLB)
4	ITLB,DTLB	Provides medium size page support. Can be runtime configured to support either 1 MB, 4 MB, 16 MB, or 64 MB pages via the ITLBCFG or DTLBCFG special registers.	4
5	ITLB,DTLB	Provides large size page support. Can be runtime configured to support either 128 MB or 256 MB pages via the ITLBCFG or DTLBCFG special registers.	4
6	ITLB,DTLB	Provides large size page support. Can be runtime configured to support either 256 MB or 512 MB pages via the ITLBCFG or DTLBCFG special registers. Note that upon processor reset, this way is set to 512 MB page size and maps the entire 4 GB address space, similarly to processors configured with Region Protection.	8
7 - 9	DTLB	4 KB page wired entries. These ways are intended for use by memory management software for entries that are not to be replaced by refill.	1 per way
Note: TLB ways 0-3 are autorefillable. The other TLB ways are not.			

1. The XPG only models a ROM at address 0xFE000000, so to place the alternate reset vector at 0x3F000000 for example, the system RAM declared in the XPG must be grown to a size of at least 1024 MB. This satisfies the XPG requirement that memory be present wherever a vector is located.

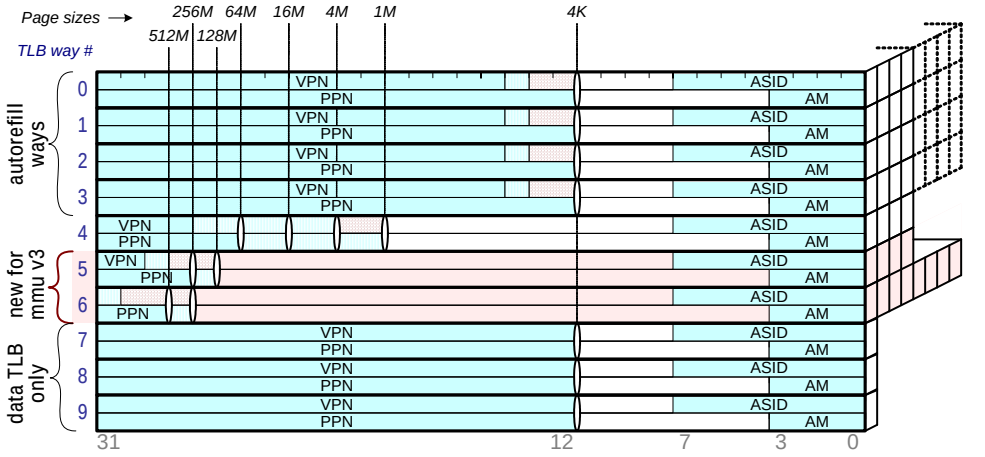


Figure 3–4. MMU TLB Layout

Page Table

The MMU page table is the data structure that controls the translations, protections, and access modes of any page represented in the page table. Any access not mapped to the TLB's wired (non autorefill) entries is looked up in the page table.

The MMU provides a total of 4 GB of mappable virtual address space. The page table is an array of 4 byte entries, each mapping a 4 KB virtual page. Therefore, the page table contains 4 GB / 4 KB = 1,048,576 page table entries (PTEs). The page table itself occupies 4 MB of virtual address space, pointed to by the PTBASE field of the PTEVADDR register. This 4 MB need not all be mapped, so the page table can take up less than 4 MB of physical space.

If large areas of virtual memory are mapped by wired TLB entries, the corresponding areas of the page table are ignored. This is illustrated in the following example, in which the latter 768 MB of virtual space are wired-mapped, thus shrinking the effective size of the page table to 4 MB - (768 MB / 1024) = 3,328 KB.

For this example, wired TLB entries are setup to provide a memory map compatible with MMU version 2, and the page table is placed at virtual address 0xCFC0_0000, at the top of remaining virtual address space. The page table spans addresses 0xCFC0_0000 to 0xCFF3_FFFF. The PTEVADDR special register is initialized to 0xCFC0_0000.

Because the page table is located in virtual address space, one part of the page table -- exactly one page of it -- maps the page table itself. Thus, the page table can be thought of as a two-level structure that resides in virtual memory. Each 4 KB page in the page table contains 1024 PTEs mapping a total of 4 MB of virtual space. One of these, the 4 KB

page that maps the entire 4 MB page table itself, is the Level-1 Page Directory Table (L1PT). Each entry in the L1PT "points to", or rather maps, a 4 KB portion of the page table that we'll call here a Level-2 PTE Table (L2PT). All the L2PTs together form the page table.

Figure 3–5 shows this example page table as seen in virtual memory.

Example code and further discussion of the MMU are also found in the *Memory Management and Protection* chapter of the *Xtensa Microprocessor Programmer's Guide*.

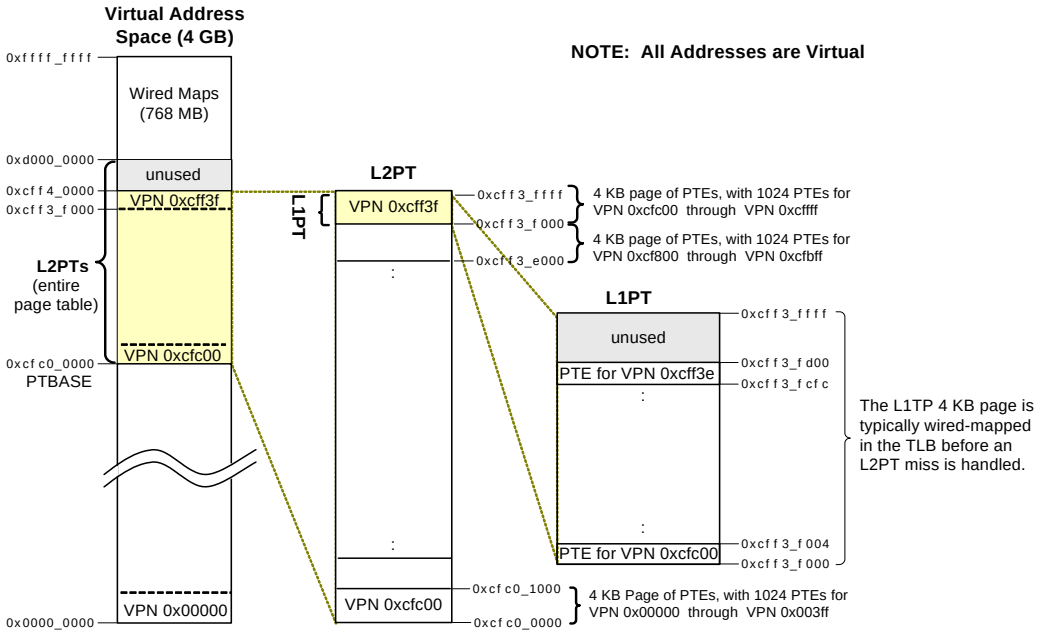


Figure 3–5. Example Page Table Organization

Each Page Table Entry (PTE) describes a virtual to physical translation for a single 4 KB page. It contains the Physical Page Number (PPN), the Ring number (RING), and the Access Mode (AM), as shown in Figure 3–6:

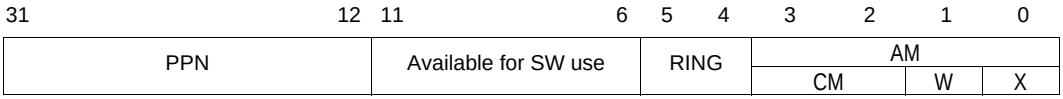


Figure 3–6. Page Table Entry - MMU with TLB

Access Modes for MMU with TLB

The Xtensa processor's MMU access modes[1] (AM) ease development of demand-paging routines in operating systems. The lower two bits restrict instruction-fetch (X) and store (W) accesses to the designated page. The upper two bits define the memory-access and cache behavior (CM). These bit fields can also be called the executable (X), writable (W), and cache mode (CM) bits.

If the access mode is marked as invalid (CM = 3 and X = 0), the appropriate exception (`InstFetchProhibitedCause`, `LoadProhibitedCause`, or `StoreProhibitedCause`) will be taken when the page is accessed. These exceptions allow the operating system to bring the required page into memory and to update the access-mode bits and the TLBs as needed. Other access modes with CM = 3 are reserved for testing (cache isolate mode) and future use.

In all other cases (CM < 3), load accesses always succeed, while the W and X bits determine the fetch and store behavior. If the page is not marked executable (X=0), an instruction fetch to that page will cause an `InstFetchProhibitedCause` exception. If the page is not marked writable (W = 0), a store operation to that page will cause a `StoreProhibitedCause` exception. The X bit allows an operating system to selectively prohibit execution from certain pages, such as for the stack and heap sections. The W bit is commonly used to support copy-on-write and to maintain a page-dirty bit in software.

Table 3–4. Access Modes for MMU with TLBs

Access Mode CM W X	Name	Fetch Behavior	Load Behavior	Store Behavior
00 0 0	Bypass cache	Exception (1)	Bypass cache	Exception (1)
00 0 1		Bypass cache	Bypass cache	Exception (1)
00 1 0		Exception (1)	Bypass cache	Bypass cache
00 1 1		Bypass cache	Bypass cache	Bypass cache
01 0 0	Write-back Cached	Exception (1)	Cached	Exception (1)
01 0 1		Cached	Cached	Exception (1)
01 1 0		Exception (1)	Cached	Write-back
01 1 1		Cached	Cached	Write-back
10 0 0	Write-through Cached	Exception (1)	Cached	Exception (1)
10 0 1		Cached	Cached	Exception (1)
10 1 0		Exception (1)	Cached	Write-through
10 1 1		Cached	Cached	Write-through

1. Access modes are referred to as "Memory Attributes" in the ISA Reference Manual.

Table 3–4. Access Modes for MMU with TLBs (continued)

Access Mode CM W X	Name	Fetch Behavior	Load Behavior	Store Behavior
11 0 0	Invalid	Exception (1)	Exception (1)	Exception (1)
11 0 1	Special	Exception (1)	Isolate	Isolate
11 1 0	Invalid	Exception (1)	Exception (1)	Exception (1)
11 1 1	Reserved (2)	Exception (1)(2)	Exception (1)(2)	Exception (1)(2)

(1) Fetches, loads, and stores will result in InstFetchProhibitedCause, LoadProhibitedCause, or StoreProhibitedCause respectively.

(2) The entry for Access Mode Value 1111 is reserved for future use. Software should not use this entry as the behavior might change.

Instruction fetches and loads to pages marked as cached, and stores to pages marked as write-back allocate a cache entry on a cache miss. Stores to pages marked as write-through do not allocate a cache entry on miss; they simply write-through to memory. Accesses to pages marked as bypass-cache will bypass the cache entirely. For processors configured without a write-back cache, the write-back access mode reverts to write-through. For processors configured without any cache memory, the cached and write-through access modes revert to bypass cache.

When an access mode attribute is set to a reserved or illegal value, memory accesses to that region cause an exception, as listed in Table 3–4.

Table 3–5. Cache Access Mode Descriptions

Access Mode	Description
Allocate	If the target address is already cached, use the cached value. If there is a cache line allocated for this address already but the value is not already in the cache, fetch the value from memory and load it into the cache. If the value is not in the cache and there is no cache line allocated for it, allocate a cache line and load the value into the cache.
No allocate	Do not allocate a line in the cache for this address. However, if the target address is already cached, use the cached value. If there is a cache line allocated for this address already but the value is not already in the cache, fetch the value from memory and load it into the cache.
Bypass	Do not use the cache.
Write-back	Write the value to the cache. The cached value is written to the appropriate memory address only when the corresponding cache line is evicted from the cache or when the processor forces the line to be written from the cache.
Write-through	Write the value to the cache and to the target memory address simultaneously.
Isolate	This mode permits direct load/store access to the cache memory and is used for manufacturing-test purposes.
Illegal	Access to an address marked as illegal causes an exception. This access mode can be used as a mechanism for protecting memory blocks.

3.2.2 Protections

Each address and data memory segment is protected through the access modes (listed in Table 3–5) stored in the relevant TLB or page table entry.

Protections

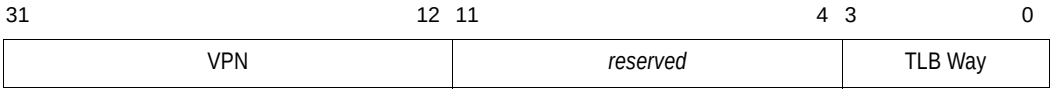
There are four protection rings and 256 ASIDs in the Xtensa processor's MMU configuration. Memory is protected through access modes, ASIDs, and rings. Privileged instructions can be executed only at Ring 0 so processor operations in this ring are protected from operations in rings 1 through 3. ASID 1 is hardwired into the Ring 0 field of the `RASID` special register, and is used to identify kernel address space. ASID 0 is reserved to mark TLB entries as invalid. This leaves 254 unique ASIDs (2 – 255) for the kernel to allocate to other processes.

Note that the kernel must also ensure that all four rings are given a unique non-zero ASID in the RASID register. In practice, given that ring 0 is assigned the kernel ASID of 1, if one ring is used for the current process and the other two rings are left unused, the two unused rings must each be given a dedicated ASID, which leaves 252 unique ASIDs for the kernel to allocate to processes (or address spaces). More processes can be supported by rotating ASID assignments.

Implementation-Specific Instruction Formats

RxTLB0, RxTLB1, WxTLB, IxTLB, and PxTLB are instructions associated with the Xtensa MMU that respectively read, write, invalidate, and probe the contents of the processor's TLBs. These MMU instructions allow an operating system to manage the TLB and are also useful for manufacturing test. For a complete listing of Xtensa MMU instructions, refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

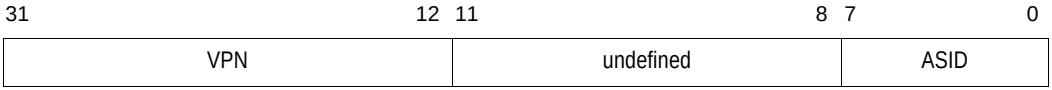
The format of the as register used for the RxTLBn, WxTLB, and IxTLB instructions is as follows. The lower 4 bits select one of the ways of the TLB targeted by the instruction (ITLB or DTLB); see Figure 3–4 for TLB layout. The upper 20 bits contain the upper bits of virtual address: the Virtual Page Number (VPN). Effectively, they can be set to the upper bits of any virtual address that could match the TLB entry to be read. Of those upper bits, only the bits that index into the selected TLB way are used, according to that way's current page size (log2(number of way entries) bits starting at bit log2(page size)). The remaining VPN bits are ignored. Bits 4 to 11 are reserved and can be either zero or the result of the relevant probe instruction, PITLB or PDTLB.



Note that operation of the relevant instructions are undefined if the TLB Way field in the as register is greater than or equal to the way count of the respective TLB.

RITLB0 / RDTLB0

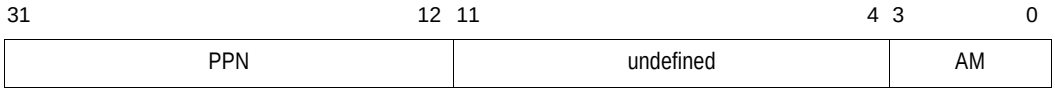
The RITLB0 and RDTLB0 instructions read the ASID and VPN out of the specified TLB entry in the following format:



If the VPN field of the relevant TLB entry is smaller than 20 bits (for the smallest page size supported by the selected TLB way), the lower bits are returned as zero.

RITLB1 / RDTLB1

The RITLB1 and RDTLB1 instructions read the PPN and Access Mode out of the specified TLB entry in the following format:

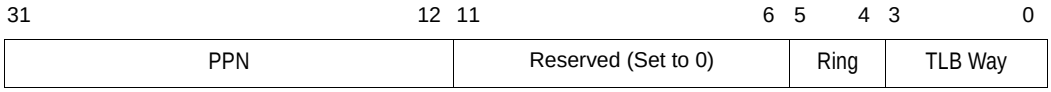


If the PPN field of the relevant TLB entry is smaller than 20 bits (for the smallest page size supported by the selected TLB way), the lower bits are returned as zero.

WITLB / WDTLB

The WITLB and WDTLB instructions write the contents of address register at to the TLB entry specified by the contents of address register as. The TLB entry's ASID field is set to the current value of one of the ASID*n* fields of the RASID register, as indexed by the Ring field of address register at.

Format for the at register:



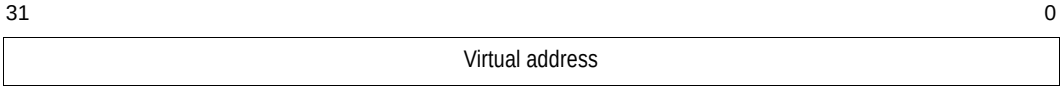
Note that special care is required if an instruction changes the MMU entry for the running code. The code must meet the following restriction. A MEMW instruction must precede the WITLB instruction. If running cached, MEMW and WITLB instructions must be in the same cache line.

IITLB / IDTLB

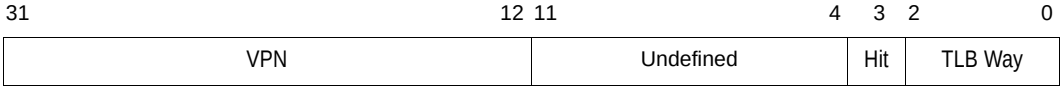
The IITLB and IDTLB instructions invalidate the contents of the TLB entry specified by the contents of address register `as`. That is, they set the entry's ASID field to zero.

PITLB / PDTLB

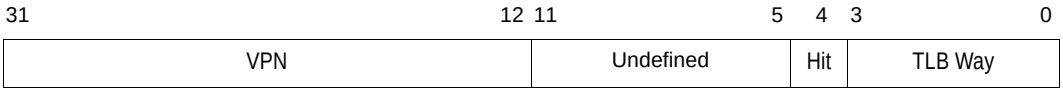
The PITLB and PDTLB instructions probe the relevant TLB for an entry that matches the virtual address specified in address register `as` (as well as the current `RASID`):



Format of the `at` register for PITLB:



Format of the `at` register for PDTLB:



If the probe succeeds, the Hit field contains a 1, otherwise it contains a 0. Also, if the probe succeeds, the value of the `at` register can be used to select the corresponding TLB entry (using the `as` register) with the RxTLBn, WxTLB, or IxTLB instruction.

3.2.3 Configuration Restrictions

There are some restrictions when using memory management options with cache or local memory. The next few sections outline these restrictions.

Cache Aliasing Conflicts

The Xtensa processor's memory caches are indexed with a virtual address and tagged with a physical address. When the *cache way size* (cache size / n for an n-way set-associative cache) greater than the minimum page size provisions are required to prohibit cache aliasing conflicts. Such conflicts can arise when a single physical memory location is mapped from multiple virtual addresses (or pages) that correspond to multiple entries in the cache for that one location.

This issue does not affect processors configured with the *Region Protection* or *Region Protection with Translation* option because the smallest "page" is a region of 512 MB, which is much larger than the cache.

This issue does not affect software that uses the default reset state of the MMU, or similarly restricts itself to using large pages only (1 MB and larger). This is the case for most RTOS's and simple runtime environments such as XTOS, unless the application specifically exercises the MMU.

Cache aliasing can, however, affect Xtensa LX7 processor cores configured with the MMU option when running software that uses 4 KB pages, whether through a page table or wired ways. This includes protected operating systems such as Linux, BSD, and others. They must manage cache-aliasing conflicts if the cache way size exceeds 4 KB.

When aliasing occurs, the virtual page number overlaps with the cache line index as shown in Figure 3–7. Multiple accesses to the same physical address through separate virtual mappings can end up in different cache entries if the overlap bits of the accessed virtual addresses differ.

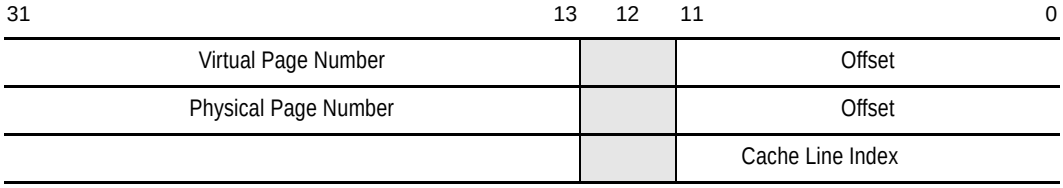


Figure 3–7. Example Cache Aliasing Overlap for an 8 KB Direct Mapped Cache

The operating system usually avoids cache-aliasing conflicts by restricting its virtual to physical mappings, by using *page coloring*, or by selectively flushing the caches. These are described here in turn.

One of the simplest approaches is to allocate memory in chunks of the cache way size, instead of in chunks of 4 KB. While simple and fast, coarser-grained memory allocation increases memory usage. Its performance advantage can also be negated on systems that page to disk, where a reduction in usable memory increases paging.

Page coloring is used as a method to mark a group of pages with different *colors*. In this case, the overlapping part of the physical or virtual address and the cache line index is used to identify the color of the page. Any request to allocate and assign a new physical page to a virtual address therefore allocates a physical page from the pool with the same color as the virtual address. This method allows multiple mappings to the same page, needed in cases such as shared memory, but it can result in higher memory fragmentation and may even run out of memory earlier if the particular memory pool is exhausted.

Another method that avoids cache-aliasing conflicts is to simply ensure that the cache contains only the contents of one mapping and that the cache sets for all other alternative mappings are flushed and invalidated prior to modifying a page. This method can only be used if no more than one mapping is valid at any time. Systems employing shared memory, for example, cannot use this approach.

Local Memory Support with MMU Option

Each local memory interface is located in physical address space. The MMU can be used to map these interfaces at any virtual address, subject to alignment and contiguity constraints: instruction RAM, instruction ROM, data RAM, and data ROM.

Specifically, a local memory interface must be naturally aligned to its size in virtual space, just as in physical space. This applies even if only a subset of the memory is mapped. In other words, the lower portion of the virtual and physical addresses used to access the local memory (the memory *index bits*) must match. Given the MMU only maps pages of 4 KB or larger, always matching the lower 12 bits of virtual and physical addresses, this restriction is only relevant for local memory interfaces larger than 4 KB.

This restriction allows local memory accesses to occur with the same timing as cache. The local memory interface may present the memory with virtual rather than physical address bits where necessary. Nevertheless, the programmer's model is that of a physically addressed local memory interface.

Note: When the memory error protection option is selected, local memories can only be configured with the byte wide ECC or Parity option if MMU is configured. Word wide memory error protection is not supported with MMU configurations.

The contiguity restriction states that two contiguous virtual pages mapped to instruction RAM or ROM, where execution might fall through from the first page to the second page, must be mapped to two contiguous physical pages (in the same order). In other words, you cannot alternately map successive virtual pages between two instruction RAMs.

Note: Execution cannot fall through from one type of instruction memory to another. These are considered three different types: instruction RAM, instruction ROM, and PIF. This restriction is not specific to the MMU option.

For speed, the XLMI interface presents the entire virtual address, not just the index bits. Therefore, software can only access XLMI using virtual addresses that completely match its physical address range (an identity map). In other words, XLMI can only be used at its configured virtual address; it cannot be remapped at other virtual addresses.

The result of accessing any local memory interface using a virtual to physical mapping that violates these constraints is undefined.

Figure 3–8 illustrates MMU support for all local memory interfaces.

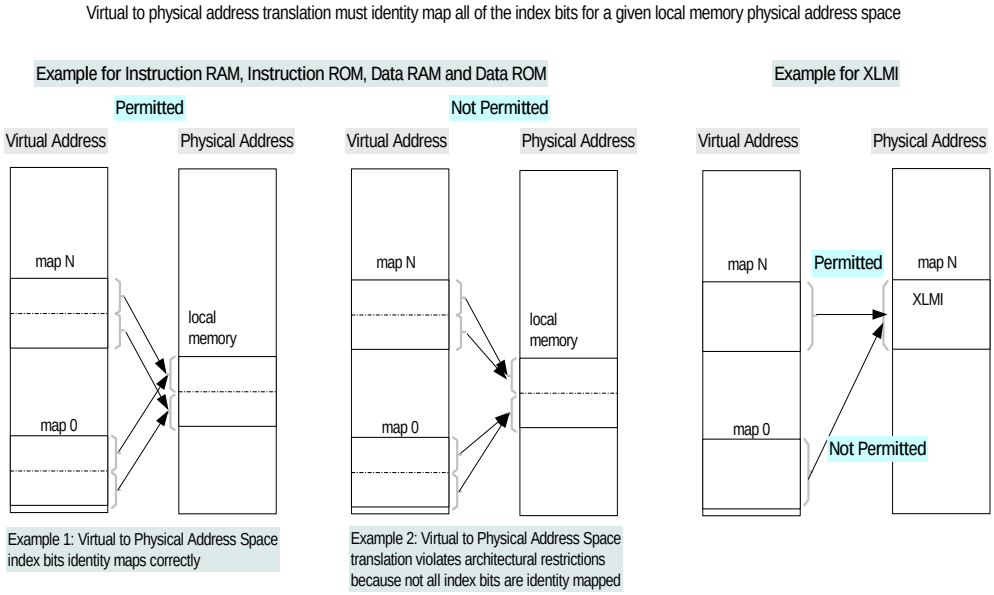


Figure 3–8. MMU Support for Local RAM, ROM and XLMI Options

Note: Example 1 in Figure 3–8 shows that virtual pages in different memory maps can map to the same pages in physical address space. Example 2 shows that more than one virtual page in one memory map cannot point to the same page in physical address space. Example 3 shows that the XLMI memory space can only be identity mapped.

A 16 KB data RAM interface, for example, puts out the lower 14 virtual address bits of memory accesses. Thus, when mapping a virtual page to a local memory, the lower 14 physical address bits must match the lower 14 virtual address bits. The upper 18 address bits can be mapped arbitrarily.

With an MMU, local memories are constrained to one of two physical address ranges as described at the beginning of Section 3.2.

MMU Exceptions

The following exceptions shown in Table 3–6 can occur only when the processor has been configured with the MMU configuration option.

Table 3–6. MMU Exceptions

MMU Exception	Cause
PrivilegeCause	Privileged instruction attempted at a current ring greater than zero.
InstFetchPrivilegeCause	Instruction fetch attempted from a page set to a lower numbered ring than the one for which fetch has privilege.
LoadStorePrivilegeCause	Load or store instruction attempted to access a page set to a lower numbered ring than the one for which it has privilege.
InstFetchProhibitedCause	Instruction fetch from page prevented by access mode (AM).
LoadProhibitedCause	Load from page prevented by access mode (AM).
StoreProhibitedCause	Store to page prevented by access mode (AM).
InstTLBMissCause	Instruction fetch finds no entry in ITLB, nor in the portion of the page table that is currently mapped in the DTLB.
LoadStoreTLBMissCause	Load or store finds no entry in DTLB, nor in the portion of the page table that is currently mapped in the DTLB.
InstTLBMultiHitCause	Instruction fetch finds multiple entries in ITLB.
LoadStoreTLBMultiHitCause	Load or store finds multiple entries in DTLB.

Loads, Stores, and Instruction Fetches

Loads and stores, whether to a local memory interface or the PIF, always use the data TLB (DTLB). Likewise, instruction fetches always use the instruction TLB (ITLB). Either way, if the access does not hit in the relevant TLB, the auto-refill mechanism attempts to load the corresponding entry from the page table using the DTLB; if successful (hit), it fills the relevant TLB and retries the access. Otherwise, a miss exception is taken.

Store Compare Conditional (S32C1I)

Xtensa configurations with an MMU-with-TLB include the Store Compare Conditional (S32C1I) instruction. The Linux kernel, for example, uses this instruction to provide synchronization among multiple tasks or exception handlers running on a single processor. This includes synchronization among user-level tasks (disabling interrupts is a privileged operation, and thus not an efficient alternative). For programs running entirely on a single processor, the S32C1I instruction can operate entirely within the cache when using the writeback cache mode, using an appropriate setting of the ATOMCTL register. When multiple processors interact using S32C1I, the ATOMCTL register must direct the S32C1I

instruction's atomic transactions outside the processor over the PIF. The PIF slave(s) that handle requests to relevant RAM must support the atomic PIF Read-Conditional-Write (RCW) transaction.

3.3 The Xtensa Memory Protection Unit (MPU)

The Memory Protection Unit (MPU) provides memory protection with a smaller footprint than the Linux-compatible MMU, but with much more flexibility and features than the Region Protection Unit.

MPU features include:

- A configurable number (16 or 32) of entries, which determine the regions
- A configurable address granularity (minimum 4 KBytes)
- 12 access-rights settings choices per region
- 200+ different memory-type choices per region, including the ability to specify internal, L1 cache behavior that is different than external (AXI or PIF) cache behavior (attributes).
- Identity map implementation: No address translation
- Runtime modifiable foreground memory map
- Static background memory map
- Unified instruction and data memory maps

Following are the major differences between the MPU and the other three memory management choices (region protection, region protection with translation, and MMU with TLB and autorefill).

- Variable Region Size:
  The Region Protection Unit uses 512 MByte regions, and the Linux-compatible MMU uses 4 KByte pages with fixed sizes. In contrast, the MPU's region sizes are runtime-modifiable in the foreground map (see Section 3.3.1).
- More Memory Types:
  The other memory management options use a 4-bit attribute code providing only six different ways to describe memory. In contrast, the MPU uses a 9-bit memory type code, providing hundreds more.

  Note that these memory types implement a dichotomy between internal (L1 Cache), and external (AXI or PIF) cache attributes. For example, you can specify a write-back (WB) L1 cache simultaneous with a write-through (WT) AXI L2 Cache. You can even specify a bypass L1 (non-cached) behavior while simultaneously caching at an external cache AXI.

- Kernel vs. User Access Protections:
  The Region Protection Unit (with and without Translation) does not provide access protection, while the Linux MMU uses a quad-concentric-ring protection scheme. In contrast, the MPU provides two levels of access privilege: kernel and user. A 4-bit access rights code implements 12 combinations of kernel vs. user privilege, crossed with read vs. write vs. execute permissions.
- Unified Instruction and Data:
  The other options implement two separate lookup tables in hardware, one each for instruction and data accesses. Whereas, the MPU implements a single lookup table, which serves both instruction and data accesses.

3.3.1 Runtime Modifiable Regions

The memory regions are defined by programming the MPU *entries*. Each entry has the following programmable fields:

- Virtual Address Start [31:*la*]
  This field contains the starting address of an MPU region.The lower address bit *la* equals 12 when the MPU is configured for 4 KByte address granularity (it is different for other granularities). All addresses starting with this address, inclusive, up to but excluding the next entry's virtual address start field fall within this MPU region.
- Access Rights [3:0]
  This field contains the access rights for this region. Of the 16 possibilities, 12 are defined. They provide salient combinations of access privilege (kernel vs. user) with access type (read vs. write vs. execute).
- Memory Type [8:0]
  This field contains the memory type applicable to this region. Over 400 choices allow the specification of parameters such as device-like memory vs. memory-like memory, shared vs. non-shared memory, internal L1 cacheability, external cacheability, PIF Attributes, AXI signaling, and ACE signaling.
- Valid
  This single-bit field selects between the foreground map and the background map for the access rights and memory type fields.

These fields are described in detail in the Memory Protection Unit Option section in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

Note that special care is required if an instruction changes the MPU entry for the running code. The code must meet the following restriction. A MEMW instruction must precede the WPTLB instruction. If running cached, MEMW and WPTLB instructions must be in the same cache line.

3.3.2 Region Determination

When an Xtensa module, such as instruction fetch, load/store, integrated DMA, or block prefetch, needs to access data at an address, it performs a *lookup*, which means it consults the MPU. During this process, the MPU compares this address against the virtual address start field programmed into each entry. The comparison outputs, taken together, indicate which entry (and hence which region) the lookup address matches. The applicable access rights, memory type, and valid fields are derived from that entry.

Each entry has a number; an MPU programming requirement is that the sequence of virtual start addresses must be monotonic with respect to the entry number. If the MPU detects that it is programmed in a non-monotonic way, it causes the Xtensa processor to take either an `InstTLBMultiHitCause` or `LoadStoreTLBMultiHitCause` exception.

The virtual address start value determines which region the lookup accesses, the lowest possible value is 32'd0 and the highest possible address is 32'hffffffff.

Figure 3–9 shows a correctly programmed MPU for an 8-entry configuration. In this figure, VAS refers to the Virtual Address Start field of a particular entry.

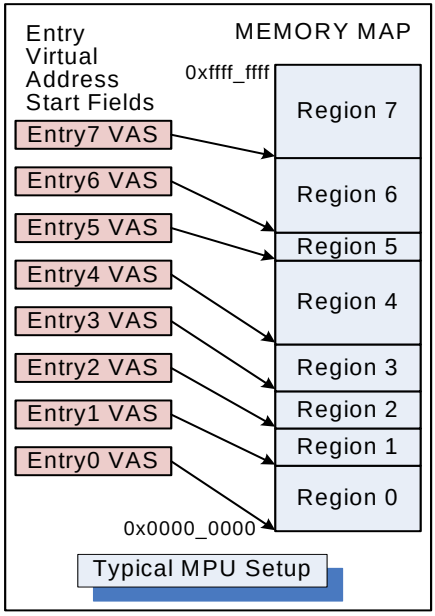


Figure 3–9. Correctly Programmed MPU

Note that in this figure, the addresses increase monotonically with respect to the Entry number.

It is possible for two or more adjacent VAS fields to contain the same address. If this is the case, the highest region eclipses the lower region containing the same virtual address start field. For example, if Entry2, Entry3, and Entry4 contain the same virtual address start field values, then the MPU behaves as if Region2, Region3, and Region4 are concatenated into one single region where the starting address equals Entry2 VAS, and the ending address equals Entry5 VAS minus 1, and the access rights, memory type, and valid bits come from Entry4.

3.3.3 Memory Type[8:0] Field Codes

In the *Xtensa Instruction Set Architecture (ISA) Reference Manual*, *The Memory Protection Unit Option* section in the *Architectural Options* chapter provides details for the memory type field codes. Refer to the subsection on *Memory Protection Unit Option Memory Type Field* for details on field values and their decoding implications. In particular, see the *Memory Protection Unit Option Memory Type* table that defines decoding of specific field values and their interpretations for the `MemoryType[8:0]` field codes.

Memory Types for L1 Data Cache

Referring to the Memory Protection Unit Option Memory Type table, it conceptually presents how the read and write allocation should occur. The read-allocation and write-allocation specifiers in the `MemoryType[8:0]` field codes are only *hints* (fully described in that table).

Similarly the Xtensa processor, when handling its internal Level-1 data cache (if an L1 data cache is configured), regards those bits as hints. The precise mechanism, for the L1 data cache, is as follows:

- Any cacheable memory type *always* read allocates, even if the "r" bit (`MemoryType[6]`) is negated.
- Any cacheable write-through memory type *never* write-allocates, even if the "w" bit (`MemoryType[5]`) is asserted.

The foregoing means that there are only three choices actually implemented for Level-1 data cache handling:

- WriteBack with ReadAllocate and WriteAllocate (WB)
- WriteBack with ReadAllocate and NoWriteAllocate (WBNA)
- WriteThrough with ReadAllocate and NoWriteAllocate (WT)

Memory Types for L1 Instruction Cache

The L1 instruction cache, if configured, is even more limited. Since the Xtensa processor never stores to its instruction cache, the writeback bit, `MemoryType[4]`, and the write allocate bit, `MemoryType[5]`, are always ignored. And, on an instruction cache miss, the L1 instruction cache always allocates, even if the read allocate bit, `MemoryType[6]`, is negated.

Reserved Memory Type Codes

Note that there are a number of reserved `MemoryType[8:0]` codes. If the MPU detects that it is programmed with one of these reserved codes, it raises an exception or denies access. More details are provided in the following Exceptions section.

For best results, the user is strongly urged to use HAL routines as defined in the *Xtensa System Software Reference Manual*.

3.3.4 Background Map

The background map is completely static; it has no state and uses only gates. It is valid at all times.

The background map provided with this processor has a single `AccessRights[3:0]` value equaling 0x7 for all addresses in the full 4Gb address space. This value provides kernel read/write/execute permission, and no user access permission.

The background map also provides a single `MemoryType[8:0]` value equaling 0x6 for all addresses in the full 4Gb address space. This value declares all memory to be of type Device (never cached), Shareable, Non-Bufferable and non-interruptible.

3.3.5 Exceptions

Following are possible exceptions generated by the MPU:

- `LoadProhibitedCause:` Causes for this exception include:
  - A load which violates read access rights
  - A load whose lookup accesses an entry programmed with a reserved access rights value
  - An S32C1I instruction that violates ONLY read access rights
  - When Block Prefetch is configured and the starting address plus length results in an address that crosses into another region
  - When an entry that is looked up is programmed with a reserved memory type value

- `StoreProhibitedCause:` Causes for this exception include:
  - A store that violates write access rights
  - A store whose lookup accesses an entry programmed with a reserved access rights value
  - An S32C1I instruction that violates write access rights, whether or not read access rights are violated
  - A DHI instruction without write access rights
  - When an entry that is looked up is programmed with a reserved memory type value
- `InstFetchProhibitedCause:` This exception happens when
  - An instruction fetch violates execute access rights
  - An instruction fetch whose lookup accesses an entry programmed with a reserved access rights value
  - When an entry is looked up is programmed with a reserved memory type value
- `InstTLBMultiHitCause` or `LoadStoreTLBMultiHitCause:` This exception occurs when a multi-hit condition in the MPU is detected (for example, during a look-up)
- `LoadStoreErrorCause:` This exception is taken when caches are configured, and at least one cache way is enabled, but the cache power is off as indicated by the CACHEADRDIS processor register. The intent of this exception is to notify the user that an L1-cacheable access requires that cache power be turned on.

3.3.6 Using the CACHEADRDIS Register

For a detailed description of the CACHEADRDIS register, refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual*, the chapter on *Architectural Options,* the subsection *Memory Protection Unit Option Register Formats* in *The Memory Protection Unit Option* section.

The CACHEADRDIS register allows the system designer to reduce cache power when the limits of cacheable addresses are known to reside outside specific 512MB regions of memory address space. The system designer can then turn on the bits of CACHEADRDIS associated with those 512MB regions which are *NOT* cacheable. Since IRAM and DRAM accesses are never cacheable, one usage model is to configure their addresses to reside in those 512MB regions where there are no cacheable addresses, thus, the associated CACHEADRDIS bits are turned on.

Each bit in the CACHEADRDIS register corresponds to a 512MB region of memory. The eight bits therein, thus, cover the full span of the available 4GB address space. If any one of those eight bits is asserted, the Load/Store unit will *NOT* turn on cache power (if caches are configured) for any accessed address in the corresponding (to that bit)

512MB region of memory. Since the Load/Store unit will not turn on cache power, any L1 cacheable load or store in that address range will result in a LoadStoreErrorCause exception as described above.

If a user desires to save power by turning on bit(s) in the CACHEADRDIS register, the following requirement must be met. If this requirement cannot be met, the CACHEADRDIS register should be left completely cleared to zero, as it is upon power-up reset.

If you turn on bit(s) in the CACHEADRDIS register, it is required that you *NOT* program an L1 *cacheable* MemoryType[8:0] (described above) into any MPU entry whose address region overlaps (fully or partially) with the 512MB address region associated with the turned-on CACHEADRDIS register bit.

If you violate the requirements by performing both of these actions:

- Turn on a bit in the CACHEADRDIS register (which corresponds to a 512MB region of address space)
- Program an MPU entry to be L1-cacheable whose address range overlaps with the same 512MB region

And, if you additionally:

- Configure an IRAM whose address range overlaps with the same 512MB region
- Place the LoadStoreErrorCause exception handler in that IRAM in the same 512MB region

then the processor can have continual repeated LoadStoreErrorCause exceptions.

There are several ways to avoid these continual LoadStoreErrorCause exceptions. One recommended way is to set the MPU entry that overlaps the 512MB region corresponding to the set CACHEADRDIS register bit to a non-cacheable memory field value as defined in the *Memory Protection Unit Option Memory Type* table in the ISA Reference Manual.

(Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual*, *The Memory Protection Unit Option* section in the *Architectural Options* chapter, the subsection on *Memory Protection Unit Option Memory Type Field* that contains the *Memory Protection Unit Option Memory Type* table, which defines decoding of specific field values and their interpretations for the `MemoryType[8:0]` field codes.)

The next section discusses how to use Hardware Abstraction Layer (HAL) routines to program the MPU. Note that if the MPU is programmed through the HAL routines, the continual LoadStoreErrorCause exceptions will not occur. The disable bit will never be set for a 512MB region that contains any part of an MPU region that has a cacheable memory type. The HAL routines for programming the MPU calculate and set CACHEADRDIS. Refer to the *Xtensa System Software Reference Manual*, chapter on *Xtensa Processor HAL*, section on *The Memory Protection Unit (MPU)* for details.

3.4 HAL and OS Support

The Xtensa processor Hardware Abstraction Layer (HAL) describes, and to some extent abstracts, the configurable and extensible portions of the Xtensa core. The Xtensa HAL enables writing software that functions properly on multiple Standard processors and Xtensa processor configurations and extensions.

Although very useful for application level code, the Xtensa HAL is particularly relevant for operating system and system software development. Such runtime code is typically sensitive to a greater number of Xtensa processor differences in configuration parameters and of the properties of any custom extensions. A runtime making full use of the HAL can generally be made to work automatically on most Xtensa processors (within the constraints of that runtime).

The Xtensa HAL comes in two forms. First is the *compile-time HAL*, a C header file that describes a processor configuration. Second is the *link-time HAL*, in the form of a configuration-specific library and a configuration independent header file defining the API. It provides processor configuration-specific information at run-time, as well as routines of general use to operating systems such as for managing caches, flushing register windows to the stack, or saving and restoring designer-defined TIE state.

Many operating systems have architecture and board-specific hardware abstraction layers also termed "HAL". The Xtensa HAL is not to be confused with these. The Xtensa HAL is OS-independent. It purely describes and abstracts features of the *processor* itself, whereas most operating system specific "HALs" abstract the features of a hardware system *outside* of the processor.

The HAL is described in detail in the *Xtensa System Software Reference Manual*.

4. Multiple Processor (MP) Features and Options

The multiple-processor (MP) option for the Xtensa architecture adds a processor ID register, break-in/break-out capabilities to the processor hardware, and optional instructions for MP synchronization. All versions of the Xtensa processor have certain optional PIF operations that enhance support for MP systems.

4.1 Processor ID Register

Some SOC designs use multiple Xtensa processors that execute from the same instruction space. The processor ID option helps software distinguish one processor from another via a `PRID` special register.

4.2 Multiple Processor On-Chip Debug

Placing multiple processors on the same IC die introduces significant complexity in SOC software debugging. The break-in/break-out option for the Xtensa Debug Module simplifies multi-core debugging. This capability enables one Xtensa processor to selectively communicate a Break to other Xtensa processors in a multiple-processor system. The option adds two interfaces and a number of control and status bits to OCD hardware.

The `BreakOut/BreakOutAck` interface communicates a break command to other processors when the processor has entered `OCDHaltMode`. The `BreakIn/BreakInAck` interface causes the processor core to take a debug interrupt if the break-in feature is enabled. These interfaces comply to the ARM® CoreSight™ Cross Trigger Interface specification, therefore they come paired with an acknowledge signal. Moreover, CoreSight's Embedded Cross-Trigger Matrix can be used to cause break-in into one Xtensa processor to come from a programmable combination of break-outs of other Xtensa cores.

The DebugStall feature compliments BreakIn/BreakOut by providing a faster method to synchronously stop and resume a multiprocessor system for debug purposes. There is no separate configuration parameter that controls this feature. It is automatically available when the OCD option is selected.

See the "Multicore Debug" section of the *Xtensa Debug Guide* for detailed information on both these features.

4.3 Multiprocessor Trace

In addition to MP debug, it is also possible to non-intrusively trace multiple processors if they are configured with the Cadence trace extraction and analysis tool, TRAX. TRAX, which is detailed in the *Xtensa Debug Guide*, is a collection of hardware and software components that provides visibility into the activity of running processors using compressed execution traces. The ability to capture real-time activity in a deployed device or prototype is particularly valuable for MP systems where there are a large number of interactions between hardware and software.

TRAX monitors the processor's Traceport and outputs changes in the processor's flow of execution. This information consists of compressed descriptions of taken branches, exceptions, and interrupts sufficient to reconstruct the processor's flow of execution. Trace collection is stopped by trigger events such as PC-match, cross-trigger signals or BreakIn/BreakOut. The latter two are particularly valuable in MP systems where each processor core is configured with TRAX. Using this feature, trace events on one processor can be linked to debug events on another processor.

4.4 Multiprocessor Synchronization Option

When multiple processors are used in a system, some sort of communication and synchronization between processors is required. (Note that multiprocessor synchronization is distinct from pipeline synchronization between instructions as represented by the `ISYNC`, `RSYNC`, `ESYNC`, and `DSYNC` instructions, despite the name similarity.) In some cases, self-synchronizing communication structures such as input and output queues are used. In other cases, a shared-memory model is used for communication. Shared-memory communication protocols require instruction-set support for synchronization because shared memory does not provide the required semantics. The Xtensa Multiprocessor Synchronization configuration option provides ISA support for shared-memory communication protocols.

4.5 Memory-Access Ordering

The Xtensa ISA requires that valid programs follow a simplified version of the Release Consistency model of memory-access ordering. The Xtensa version of Release Consistency is adapted from *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors* by Gharachorlo et. al. in the Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990, from which the following three definitions are directly borrowed:

- A load by processor *i* is considered *performed with respect to processor k* at a point in time when the issuing of a store to the same address by processor k cannot affect the value returned by the load.

- A store by processor i is considered *performed with respect to processor k* at a point in time when an issued load to the same address by processor k returns the value defined by this store (or a subsequent store to the same location).
- An access is *performed* when it is performed with respect to all processors.

Using these definitions, the Xtensa processor places the following requirements on memory access:

- Before an ordinary load or store access is allowed to perform with respect to any other processor, all previous *acquire* accesses must be performed, and
- Before a *release* access is allowed to perform with respect to any other processor, all previous ordinary load, store, acquire, and release accesses must be performed, and
- Before an acquire is allowed to perform with respect to any other processor, all previous acquire accesses must be performed.

In the context of non-coherent caches, the above definitions are extended as follows:

- When writing to non-coherent writeback-cached memory, a "store" is not simply a processor store operation, but instead includes both a processor store operation and subsequent writeback operation(s) to all corresponding cache lines. (Programmatically, whatever normally must follow a store, in this case must follow the writeback that follows the corresponding store.)
- When reading from non-coherent cached memory, a "load" is not simply a processor load operation, but instead includes both a processor load operation and previous invalidate operation(s) to all corresponding cache lines. (Programmatically, whatever normally must precede a load, in this case must precede the invalidate that precedes the corresponding load.)

In practice, this means, for example, that if you store to writeback-cached memory, then writeback, then MEMW, then some bypass access, the store and the bypass access are ordered at processor interfaces (and in the system assuming it properly preserves ordering).

Future Xtensa processor implementations may adopt stricter memory orderings for simplicity. Software programs, however, should not rely on any stricter memory ordering semantics than those specified here.

4.5.1 Architectural Additions

Table 4–7 shows the Multiprocessor Synchronization option's architectural additions.

Table 4–7. Multiprocessor Synchronization Option Instruction Additions

Instruction	Format	Definition
L32AI	RRI8	Load 32-bit Acquire (8-bit shifted offset) This load will perform before any subsequent loads, stores, or acquires are performed. It is typically used to test the synchronization variable protecting a mutual-exclusion region (for example, to acquire a lock).
S32RI	RRI8	Store 32-bit Release (8-bit shifted offset) All prior loads, stores, acquires, and releases will be performed before this store is performed. It is typically used to write a synchronization variable to indicate that this processor is no longer in a mutual-exclusion region (for example, to release a lock).

4.5.2 Inter-Processor Communication with the L32AI and S32RI Instructions

The 32-bit load and store instructions L32AI and S32RI add acquire and release semantics to the Xtensa processor. These instructions are useful for controlling the ordering of memory references in multiprocessor systems, where different memory locations can be used for synchronization and data and precise ordering between synchronization references must be maintained. Other Xtensa load and store instructions may be executed by processor implementations in any order that produces the same uniprocessor result.

L32AI is used to load a synchronization variable. This load will be performed before any subsequent load, store, acquire, or release is begun. This characteristic ensures that subsequent loads and stores do not see or modify data that is protected by the synchronization variable.

S32RI is used to store to a synchronization variable. This store will not begin until all previous loads, stores, acquires, or releases are performed. This characteristic ensures that any loads of the synchronization variable that see the new value will also find all protected data available as well.

Consider the following example:

```
volatile uint incount = 0;

volatile uint outcount = 0;

const uint bsize = 8;

data_t buffer[bsize];

void producer (uint n)

{

        for (uint i = 0; i < n; i += 1) {

                data_t d = newdata();// produce next datum

                while (outcount == i - bsize);// wait for room

                buffer[i % bsize] = d;// put data in buffer

                incount = i+1;   // signal data is ready

        }

}

void consumer (uint n)

{

        for (uint i = 0; i < n; i += 1) {

                while (incount == i);// wait for data

                data_t d = buffer[i % bsize];// read next datum

                outcount = i+1;  // signal data read

                usedata (d);     // use datum

        }

}
```

Here, `incount` and `outcount` are synchronization variables, and `buffer` is a shared data variable. The `producer`'s writes to `incount` and `consumer`'s writes to `outcount` must use `S32RI` and `producer`'s reads of `outcount` and `consumer`'s reads of `incount` must use `L32AI` to ensure the proper ordering of loads and stores. If `producer`'s write to `incount` were performed with a simple `S32I`, the processor or memory system might reorder the write to `buffer` after the write to `incount`, thereby allowing `consumer` to see the wrong data. Similarly, if `consumer`'s read of `incount` were done with a simple `L32I`, the processor or memory system might reorder the read to `buffer` before the read of `incount`, also causing `consumer` to see the wrong data.

4.6 Conditional Store Option

A multiprocessor system can require mutual exclusion, which cannot easily be programmed using the Multiprocessor Synchronization Option. The Conditional Store Option adds that capability with one instruction (S32C1I) that atomically stores to a memory location only if the current value stored in that location is the expected one. A state register (SCOMPARE1) is also added to provide the additional operand required.

Note: The Conditional Store operation in the processor is implemented inside the processor for cached write-back regions of memory as well as for local data memories, and requires no additional functionality in the memory controller or bus interface. However, to perform Conditional Store operations to a cached write-through region of memory, or to an uncached region of memory, the bus connecting the processor and the shared local memory are required to support atomic read-compare-write operations. This can be accomplished with something as simple as a lock mechanism on the associated memory arbiter; in particular the Cadence-supplied AHB and AXI bridges support the Conditional Store operation. For PIF-attached memories, the associated memory controller must support the atomic read-compare-write operation.

4.6.1 Architectural Additions

Table 4–8 and Table 4–9 show the Conditional Store option's architectural additions.

Table 4–8. Conditional Store Option Processor-State Additions[1]

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
ATOMCTL	1	6	Atomic Operation Control	HW	99
SCOMPARE1	1	32	Expected Data Value for S32C1I Instruction	R/W	12

1. Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions.

Table 4–9. Conditional Store Option Instruction Additions

Instruction	Format	Definition
S32C1I	RRI8	Store 32-Bit Compare Conditional Stores to a location only if the location contains the value in the SCOMPARE1 register. The comparison of the old value and the store, if equal, are expected to be atomic. For processor configurations without an AXI4 bridge, the instruction always returns the old value of the memory location. For processor configurations that have anAXI4 bridge, the old value of the memory location is returned most of the time; however in some cases ~SCOMPARE1 is returned, indicating an atomic update failure. Note that this kind of failure with AXI4 can only happen when the ATOMCTL register is set to External.

4.6.2 Exclusive Access with the S32C1I Instruction

L32A and S32RI allow inter-processor communication, as in the producer-consumer example shown in Section 4.5.2 on page 44 (barrier synchronization is another example). However, these instructions are not efficient for guaranteeing exclusive access to data (for example, locks). Some systems may provide efficient, tailored, application-specific exclusion support. When this is not appropriate, the Xtensa ISA provides another general-purpose mechanism for atomic updates of memory-based synchronization variables that can be used for exclusion algorithms. The S32C1I instruction tries to store to a location if the location contains the value in the SCOMPARE1 register. The comparison of the old value and the store, if equal, are expected to be atomic. For processor configurations that do not have an AXI4 bridge, S32C1I always returns the old value of the memory location, thus it looks like both a load and a store. This allows the program to determine if the store succeeded, and if not, it can use the new value as the comparison for the next S32C1I.

If the processor configuration has an AXI4 bridge, S32C1I should behave in the same way most of the time. However in some cases, if the ATOMCTL register is set to External and if the external exclusive access monitor indicates an atomic update failure, it may return ~SCOMPARE1. Note that the memory location is not updated in this case.

For example, an atomic increment could be done as follows:

```
      movi      a5,0xFFFFFFFF
looptop:
      l32i      a3, a2, 0      // current value of memory
loop:
      wsr       a3, scompare1  // put current value in SCOMPARE1
      mov       a4, a3         // save for comparison
      addi      a3, a3, 1      // increment value
      s32c1i    a3, a2, 0      // store new value if memory
                               // still contains SCOMPARE1
      xor       a6, a4, a5     // generate ~SCOMPARE1
      beq       a3, a6, looptop// if atomic store failed
                               // load current value again
      bne       a3, a4, loop   // if value changed, try again
```

Semaphores and other exclusion operations are equally simple to create using the S32C1I instruction.

S32C1I instructions may target cached, cache-bypass, and data RAM memory locations. S32C1I instructions are not permitted to access memory addresses in data ROM, instruction memory or the address region allocated to the XLMI port. Attempts to direct the S32C1I at these addresses will cause an exception.

For single processor systems where S32C1I is used to enforce atomicity among multiple threads, it is possible to implement it in the Data Cache. This implementation choice will be considerably faster and not require additional hardware capability outside the processor.

For multi-processor systems which use S32C1I to enforce atomicity between threads on different processors, system level hardware is required to respond to the special PIF bus transaction called Read-Conditional-Write (RCW).

To accommodate both of these design goals, the ATOMCTL register is used to control how the S32C1I instruction interacts with the data cache and the PIF bus. The following behaviors are supported:

1. When ATOMCTL=Internal communication only among threads on a single processor is supported. In privileged systems of this sort, the ATOMCTL register can be set so that unprivileged code cannot cause the special PIF bus transaction, Read-Conditional-Write (RCW), in case it is not supported by external hardware.
2. When ATOMCTL=External, communication between threads on separate processors is supported if external hardware exists to support the Read-Conditional-Write (RCW) special PIF bus transaction.
3. When ATOMCTL=Exception, the S32C1I instruction will cause an exception so that system software may emulate its behavior.

Note that the ATOMCTL register does not affect the behavior of S32C1I to data RAM in any way. S32C1I to data RAM will perform an atomic transaction without any PIF transaction. Similarly, inbound PIF RCW transactions to data RAM will perform atomic transactions. See Section 16.9 “Data Cache Conditional Store Transactions” on page 268 for S32C1I operations on cached memory.

There are many possible atomic memory primitives. `S32C1I` was chosen for the Xtensa ISA because it can easily synthesize all other primitives that operate on a single memory location. Many other primitives (for example, test and set, or fetch and add) are not as universal. Only primitives that operate on multiple memory locations are more powerful than `S32C1I`.

4.6.3 Memory Ordering and the S32C1I Instruction

With regard to the memory ordering defined for `L32AI` and `S32RI` in Section 4.5.2 on page 44, `S32C1I` plays the role of both acquire and release. That is, before the atomic pair of memory accesses can perform, all ordinary loads, stores, acquires, and releases must have performed. In addition, before any following ordinary load, store, acquire, or release can be allowed to perform, the atomic pair of the `S32C1I` must have performed. This allows the Conditional Store to make atomic changes to variables with ordering requirements, such as the counts discussed in the example in Section 4.5.2 on page 44.

4.7 Exclusive Access Option

The exclusive access option allows multiple masters in a system to access a memory location in Xtensa processor's data RAMs atomically. It also allows multiple threads of a processor to access a memory location in its data RAMs or data cache atomically. The conditional store synchronization option and the exclusive access synchronization option are mutually exclusive. Following is a typical exclusive access operation where the master performs:

1. A load exclusive instruction loads data from a memory location.
2. Some computation on the loaded data is performed.
3. An exclusive store with the updated value is stored to the same memory location.
   - The exclusive store succeeds if no other master has performed a store to that location.
   - The exclusive store fails if the location was updated by another master since it was read.

The exclusive instructions and the associated exclusive monitor will only allow a memory location to be updated if no other master has performed a store to that same location following an exclusive load.

4.7.1 Exclusive Access Instructions

The exclusive access option adds the `L32EX`, `S32EX`, `CLREX` and `GETEX` instructions to achieve atomicity.

Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for detailed specifications of these instructions.

In addition, two new micro-architecture state registers, `EXCELMON` and `EXCLRES`, are provided to maintain the necessary state required for an atomic transaction. `EXCLRES` is referred as `ATOMCTL[8]`in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

Following is an example instruction sequence to demonstrate how a typical exclusive transaction is performed:

1. When a master wishes to perform an atomic transaction, it first issues an `L32EX` instruction.
2. After the `L32EX` instruction commits, the `EXCLMON` state is set which indicates that an exclusive monitor has been activated.
3. The master then issues an `S32EX` instruction to update that same location.

If the memory value is unchanged since the L32EX was executed, the S32EX is successful, EXCLRES (exclusive result) is set to 1 and the location is updated.

If a different master updates the memory in between the L32EX and the S32EX, the memory location is not updated and EXCLRES is cleared.

In the event that the S32EX instruction fails (indicated by EXCLRES), the master must retry the L32EX/S32EX sequence until success is achieved.

Overall, any combination of semaphores and other exclusion operations can be created using the L32EX, S32EX,and GETEX instructions. The GETEX instruction allows the processor to read the EXCLRES (exclusive result) register and CLREX clears EXCLMON (exclusive monitor).

L32EX and S32EX instructions may target cached, cache-bypass (AXI master port), and data RAM memory locations. These instructions are not permitted to access memory addresses in data ROM, instruction memory, or the address region allocated to the XLMI port. Attempts to direct the exclusive access at these addresses will cause an exception.

For single processor systems where exclusive access is used to enforce atomicity among multiple threads, it is possible to implement it in the data cache. Support for multi-processor systems that use exclusive access to enforce atomicity between threads on different processors is accomplished using the AXI bus transactions, exclusive read request and exclusive write request.

4.7.2 Memory Ordering and the L32EX and S32EX Instructions

L32EX and S32EX instructions wait for all ordinary and exclusive stores to complete before they are executed.

4.7.3 AXI Master (Processor Outbound) Exclusive Operations

An Xtensa processor can issue exclusive operations on its outbound AXI master port to perform atomic access on a remote address.

Outbound Exclusive Load

The Xtensa processor does not allocate monitors for outbound (AXI master) exclusive access transactions and expects the targeted AXI slave agent to handle the monitoring.

L32EX instructions targeting the system memory will appear as exclusive load transactions on AXI. All exclusive loads on AXI are expected to receive an EXOKAY response, which indicates that exclusive accesses is supported for the address of the transaction.

If exclusive accesses are not supported by the targeted AXI slave, then the transaction is expected to get an OKAY (failed) response, which will result in the Xtensa processor taking an `ExclusiveErrorCause` exception.

Outbound Exclusive Store

The behavior of `S32EX` instructions depends on the state of the `EXCLMON` register. Normally, the preceding `L32EX` instruction would have set the `EXCLMON` register. But if the `EXCLMON` register has been cleared with a `CLREX` instruction, the exclusive store will be suppressed. If `EXCLMON` is set, the `S32EX` instruction is completed and an exclusive store transaction is started on AXI. The pipeline does not wait for a response and will continue to execute instructions. However, any load/store instruction bound for the AXI is held by the pipeline until the response for the exclusive store is received from AXI.

4.7.4 AXI Slave (Inbound) Exclusive Operations

An Xtensa processor core configured with an inbound AXI port can accept inbound exclusive requests from other AXI masters in the system that target its local data RAMs. This allows applications running on multiple processors to share a semaphore in the Xtensa processor's data RAM.

4.7.5 Monitors for Exclusive Access

The exclusive monitors are a hardware structure that keep track of the status of every pending atomic operation. Each monitor describes a continuous range of bytes from masters that are requesting atomicity.

In term of usage, the monitors can be classified into two categories: monitors for external access (AXI slave) and monitors for load/store access from the local pipeline to data cache and local data RAM.

Each monitor keeps track of the following information once activated by an exclusive read access:

- **Master ID**: The master AXI Read ID (ARID) for the incoming exclusive request.
- **Physical address**: The physical address of the incoming exclusive request.
- **Size**: The size of the incoming exclusive request.

4.7.6 Exclusive Access Transaction Support

Only exclusive accesses to the data RAM are supported. The Xtensa processor will respond with OKAY (fail) instead of EXOKAY in the following cases:

- The exclusive load or store access targets InstRAM or InstROM

- The size of the exclusive load or store access is greater than the maximum size allowed by the monitor, which is the AXI slave port data width
- If exclusive access is not configured but AXI is configured

Note that if exclusive write WSTRB does not match with exclusive read ARSIZE in all of the above cases, the store will not update any memory.

Monitors for Inbound Accesses

Every AXI master that can potentially access through Xtensa's inbound slave port is assigned a monitor. The number of monitors can be configured to be 1, 2, 4, 8, or 16.

When an inbound load exclusive transaction occurs, a monitor is allocated. The Agent ID/ AXI ID, address, and size are all recorded in a monitor and an EXOKAY result is returned. The allocation first tries to find an unused monitor. If none is available, it randomly selects from one of the monitors currently in use. If the master was holding any previous monitor, that monitor will be cleared.

Note: Having more AXI masters than the number of configured monitors can degrade performance in cases where all masters are simultaneously doing exclusive accesses.

When a store exclusive arrives, all monitors are checked. If a valid monitor matches the AXI ID (ARID), physical address, and size, then the store is completed and the matching monitor is cleared. EXOKAY is returned on AXI. For all other monitors, the store exclusive transaction is treated like an ordinary store and a match causes the associated monitor to be cleared. If the store exclusive does not match any monitor, the store is not done, OKAY is returned, and no monitors are cleared.

Ordinary inbound loads do not affect the state of the monitors. When an ordinary inbound store arrives, all monitors are checked for a match to the address/size combination. Specifically, each byte of the ordinary store is checked to see if it falls within one of the address/size descriptors. If any byte matches, the corresponding monitor must be cleared.

Monitor for Pipeline Access

The exclusive monitor can also receive exclusive access requests from the local processor. For this purpose, there is a single, dedicated monitor for accesses from the local processor.

This single monitor is in addition to the number of external monitors specified in the configuration. Pipeline accesses have no master ID. The monitor is otherwise identical to the general set. Ordinary stores from the local processor have the same effect on monitors as ordinary stores from inbound AXI.

iDMA Accesses

The iDMA is unable to do exclusive accesses to the local memory, but iDMA stores affect the global monitors in the same way as any other stores from the pipeline or inbound.

Monitor Contamination

Each monitor describes a continuous range of bytes that is accessed by a master for exclusive read or exclusive write. If any bytes in that range are written by any master using a non-exclusive store or a successful exclusive store, the exclusivity is lost. This type of a write to a memory location is called *contamination*. The Xtensa processor uses the data RAM access width, not the size of the exclusive access request, as the size of the contamination window.

For example, assume a monitor is monitoring address 0 with byte size 4. A write to address 6 with byte size 1 will contaminate and invalidate the monitor if the data RAM access width is greater than 4 bytes.

Once a monitor is contaminated, any subsequent exclusive store will fail with an OKAY response.

4.7.7 Access Conflict Between Inbound Exclusive Read-Write and Exclusive Load-Store

If an exclusive load or store to data RAM matches the address of an inbound exclusive read or write, it is a hazard situation. The Xtensa processor resolves these hazards by allowing only one to continue, and the other one waits for the first one to complete.

4.8 Inbound Processor Interface (PIF) Operations

An Xtensa processor core can optionally accept inbound requests from the PIF that access the local memories. The processor must have a PIF for the inbound-PIF Request option to be enabled. The processor must also have at least one instruction RAM or data RAM to be compatible with this option. Inbound-PIF requests are configured separately for each configured local memory.

Using inbound-PIF operations, other processors can write to and read from this processor's local memories (provided that each processor's PIF is connected to a common bus). This feature can be used to implement relatively "glue-less" MP communications. In addition to read and write operations, inbound-PIF operations also include the ability to perform conditional stores (inbound `S32C1I`) or exclusive access for applications requiring mutual exclusion implemented in the data RAM.

4.8.1 Simultaneous Memory-Access and Inbound-PIF Operations

Core processor instructions and inbound-PIF transfers can access local data memories simultaneously, provided that the addresses are to different data RAMs or data RAM banks. This feature can significantly boost system performance.

5. Xtensa Software Development Tools and Environment

Xtensa processors are supported by industry-leading software development tools including a complete C development tool chain, integrated development environment, and other debugging and development resources. Using the Xtensa software development tools, you can compile, run, and debug your C/C++ and assembly application on the single-processor Instruction Set Simulator (ISS), the multiple-processor Xtensa SystemC (XTSC) package and Xtensa Modeling Protocol (XTMP) simulation environment, or actual Xtensa hardware. XTSC and XTMP can be used to model and display application performance. Xtensa processors have full support from leading RTOS vendors with automatic RTOS support for each unique Xtensa processor configuration.

5.1 Xtensa Xplorer IDE

Xtensa Xplorer serves as a cockpit for multiple-processor SOC hardware and software design. Xtensa Xplorer integrates software development, processor optimization, and multiple-processor SOC architecture tools into one common design environment. It also integrates SOC simulation and analysis tools. Xtensa Xplorer is a visual environment with a host of automation tools that makes creating Xtensa processor-based SOC hardware and software much easier.

Xtensa Xplorer also serves as the gateway to the web-hosted Xtensa Processor Generator (XPG). Complete processor creation builds are invoked through Xplorer and downloaded to your desktop from within Xplorer.

Xtensa Xplorer is particularly useful for the development of TIE instructions that maximize performance for a particular application. Different Xtensa processor and TIE configurations can be saved, profiled against the target C/C++ software, and compared. Xtensa Xplorer even includes automated graphing tools that create spreadsheet-style comparison charts of performance.

Xtensa Xplorer dramatically accelerates the processor optimization cycle by providing an automated, visual means of profiling and comparing different processor configurations. With Xtensa Xplorer, it takes only a few minutes to try out a new instruction and get valuable feedback, right at the designer’s desktop. Then, after all the instructions are evaluated, it takes approximately one to two hours for the Xtensa Processor Generator to create a new, customized version of the Xtensa processor, including all of the custom instructions plus the associated software environment. This makes it easy for designers to try many different options.

Xtensa Xplorer helps bridge the hardware-software design gap for extended instructions by providing a context-sensitive TIE code editor, a TIE-instruction-aware debugger, and a gate-count estimator. The gate-count estimator gives real-time feedback to software designers unfamiliar with hardware development, allowing them to explore various embedded processor instruction set architectures while receiving immediate feedback on the cost implications of their extended-instruction choices.

The TIE Port and Queue Wizard offers a simple GUI interface for creating ports and queue without the need to write TIE. The output is a TIE file (and test environment) that gets incorporated into the normal flow.

Xtensa Xplorer supports both single-processor and multiple-processor SOC (MPSOC) designs based on Xtensa processors. It facilitates MPSOC development with tools for build management, profiling, batch building, and both synchronous and independent MP debugging using multiprocessor simulators or hardware targets.

Xtensa Xplorer also provides a unified environment for:

- C/C++ application software development
- C/C++ program debugging
- Code profiling and visualization
- Real-time trace through TRAX
- SOC configuration management
- Vectorization optimization (Vectorization Assistant)

Xtensa Xplorer automatically generates and manages makefiles. It provides a graphical debugger and extensive performance visualization tools.

Xtensa Xplorer can automatically drive designer-directed exploration of different processor configurations including analysis of application performance under different cache configurations. Xplorer, which is based on the Eclipse universal tool platform, has plug-ins available from third parties that support automatic integration with most source-code management systems.

5.2 Xtensa C and C++ Compiler (XCC)

The Xtensa C and C++ Compiler (XCC) is an optimized compiler for the Xtensa processor. Operation of XCC is similar to operation of standard GNU C and C++ compilers (GCC). XCC provides superior code execution performance and reduced code size compared to GCC through improved optimization and code generation technology.

5.2.1 Profiling Feedback

XCC can use run-time statistics to improve its own results. Many tradeoffs made by a compiler when optimizing code are affected by the direction of branches and the relative execution frequency of different source functions. Profiling information allows XCC to better estimate code behavior.

To use this feature, you first compile your application using a compiler flag *that directs* the compiler to instrument your application so that you can generate profiling statistics. Then you run your application using a representative data set or data sets to generate the statistics. The closer the data sets match the data expected by your actual application, the better the compiler will do. Your application can be run in the ISS or on a hardware system connected to the Xtensa Xplorer debugger or to the gdb debugger via the on-chip debug (OCD) connection. After generating the feedback files with the profiling statistics, you recompile your application using another compiler flag that optimizes the code using the profiling information. The other compiler-optimization flags and the source code should not be changed without regenerating the feedback files. If you attempt to use obsolete feedback files, the compiler will warn you.

Feedback can significantly improve run-time performance, but can have an even greater influence on code size because it allows the compiler to automatically decide which regions of code to compile for speed and which regions to compile for compactness.

5.2.2 Interprocedural Analysis (IPA)

Interprocedural analysis (IPA) allows the compiler to evaluate multiple C or C++ source files together as a group during optimization and thus allows the compiler to improve performance when compiling for speed and to reduce code size when compiling for size. To invoke IPA, you simply use the `-ipa` flag during both the compile and link steps. Individual source files can still be compiled separately. Behind the scenes, IPA delays optimization until the link step.

5.2.3 Vectorization Assistant

On processors that support SIMD instructions, such as ConnX BBE16, Vectra, and ConnX D2, the compiler is able to automatically detect opportunities for vectorization from standard C code. The dependence analysis of the compiler analyzes both inner and outer loops to see if they can safely be executed in parallel and to see if memory accesses are stride-1. For appropriate loops, the compiler replaces the scalar operations in the loop with corresponding vector ones. A Vectorization Assistant GUI is available via Xtensa Xplorer to help you navigate your code and re-write lines that prevent the compiler from automatically vectorizing important loops.

5.3 GNU Software Development Tools

Many of the Xtensa software development tools are built around the Free Software Foundation's GNU tools. The following components are included in the Xtensa software-development tool suite:

- Assembler
- Linker
- Symbolic debugger
- Binary utilities
- Profiler
- Standard libraries

5.3.1 Assembler

The GNU assembler translates Xtensa assembly source files into machine code using a standard ELF object file format. Besides the standard features of the GNU assembler, including a rich set of directives and a built-in macro processor, there are a number of special assembler features for Xtensa processors. For example, the assembler can:

- Automatically align branch targets to improve performance
- Translate pseudo-instructions into configuration-specific opcodes
- Automatically convert instructions with out-of-range immediate operands into equivalent sequences of valid instructions
- Use its scheduler to rearrange code and to bundle FLIX operations

The Cadence version of the GNU assembler also includes a dynamic plug-in capability to make it easy to add designer-defined TIE instructions or switch Xtensa processor configurations.

5.3.2 Linker

The GNU linker combines object files and libraries into executable programs. Its behavior is controlled by linker scripts that give you flexible control over where the code and data are placed into memory. The Xtensa version of the GNU linker can also remove redundant literal values and optimize calls across object files.

Cadence has introduced the concept of linker support packages (LSPs) as a way of organizing the linker scripts and runtime libraries associated with different target environments. During development, embedded software is often run using a variety of environments—functional testing with simulation, hardware verification, emulation boards,

etc.—and these environments may require different memory layouts and runtime libraries. LSPs provide an easy way to organize support for these different runtime environments.

Besides providing several default LSPs, Cadence supplies a tool to simplify the process of creating new linker scripts. This tool allows you to specify a system memory map in a simple format without needing to learn how to write linker scripts directly. Of course, if you need the full power and control of custom linker scripts, you can also create new LSPs with your own linker scripts.

5.3.3 Debugger

The GNU debugger (GDB) provides sophisticated symbolic debugging for C, C++, and assembly-language programs. You can access GDB directly with a command-line interface, or you can control GDB via the Xtensa Xplorer integrated development environment (IDE). GDB provides many features to control the execution of a program and to analyze its behavior. These features include:

- Breakpoints (with or without conditions)
- Watchpoints (to stop whenever the value of an expression changes)
- Single-stepping (by either source lines or machine instructions)
- Stack backtracing
- Examining memory (in terms of either raw values or high-level data structures)
- Viewing processor registers and other processor state
- Integrated disassembler

5.3.4 Binary Utilities

The GNU binary utilities are a set of tools that manipulate and analyze binary object files. These tools are an essential part of the software development environment. They include tools for:

- Building and modifying archive libraries of object files
- Transforming object files in various ways, such as stripping debugging information or translating to a different binary format (for example, S-records)
- Extracting information from object files, including symbol tables, size information, ELF headers, and disassembled code

5.3.5 Profiler

Profiling allows you to tune a program's performance by locating portions of code that consume the most execution time, where you'll want to focus your optimization efforts for maximum effect. You can tune Xtensa code using the traditional approach to software tuning (by optimizing the C, C++, and assembly language code) or with TIE. Tuning the processor with TIE is likely to produce much faster code.

Profiling is supported in the Xtensa Xplorer IDE or on the command line using the gprof GNU profiler. The profiler uses information collected during the program's execution and summarizes it for analysis. Several output forms are available. The flat profile output shows the number of invocations and time spent in every function. The call graph output shows the number of times a function is invoked from various call sites, the number of times it calls other functions, and estimates of the execution time attributable to each function and its descendants.

Profiling is supported using either the Xtensa Instruction Set Simulator (ISS) or on actual hardware. On actual hardware systems, profiling relies on program-counter sampling to generate statistically accurate approximations. The generated data is automatically retrieved from the target using Xtensa Xplorer or gdb.

For profiling with the cycle-accurate ISS, Cadence uses a customized approach that collects profile data unobtrusively as every instruction is executed, which is more accurate than sampling the program counter. In addition to execution-time profiling, the Xtensa ISS collects information about other events such as cache misses and pipeline interlocks. The profiler can then analyze these events. This feature provides an additional level of detail to better understand a program's performance.

5.3.6 XMP for Shared Memory Programming

The Cadence XMP environment provides a methodology and library for programming symmetric and asymmetric shared memory processors. Shared memory systems are specified in Xplorer. The user partitions memory between private memory for each processor and a segment for holding shared data. The library provides software routines for synchronization, barriers, and memory allocation. Xplorer provides out-of-the-box support for building the binaries and then simulating, profiling, and debugging the resultant systems.

5.3.7 Standard Libraries

Xtensa's software-development environment includes a complete set of standard libraries for use with your application programs.

- Standard C++ Library

  This is the same library used by the popular GNU C++ compiler.
- C Library (libc):

Cadence offers three choices of C libraries: a proprietary C library, xclib, Red Hat's newlib library and uClibc from CodePoet Consultation. The newlib and xclib libraries provide higher performance, while uClibc is geared towards minimizing code size. Xclib also provides a smaller code size than newlib, but partially achieves that by using less sophisticated algorithms for C functions such as memory allocation. Xclib is not subject to any open source licenses. Math libraries (libm) is a complete set of standard mathematical functions. There are three versions of the libraries corresponding to the three different C libraries.

- Compiler Support Library (libgcc)

  Runtime support functions for use by the compiler. This library comes from the GNU compiler, but Cadence has added hand-optimized, configuration-specific implementations of the integer multiply and divide functions as well as floating-point emulation functions.

  Note that some third-party operating systems may provide their own implementations of the above libraries.

5.4 The xt-trace Tool

The Xtensa Traceport is a hardware configuration option. Its signals reflect the processor's internal state during operation. The xt-trace software tool converts raw trace information from the Traceport into a stream of assembly instructions. It also provides varying levels of information about the processor's state. For Xtensa configurations that include the Traceport option, a separate configuration option called TRAX is available that provides increased visibility into the activity of the processor core.

5.5 TRAX

TRAX is a collection of hardware and software that provides increased visibility into the activity of running Xtensa processors using compressed execution traces. TRAX is particularly useful for software and system development and debugging.

TRAX-PC is the first generation TRAX product. It provides program-execution trace capabilities. Using a circular TraceRAM and a simple set of triggers, TRAX-PC captures and compresses the processor's flow of execution around a trigger point of interest in real time and captured for later analysis. A software tool converts captured traces into an annotated sequence of assembly instructions.

5.6 Simulation Environment

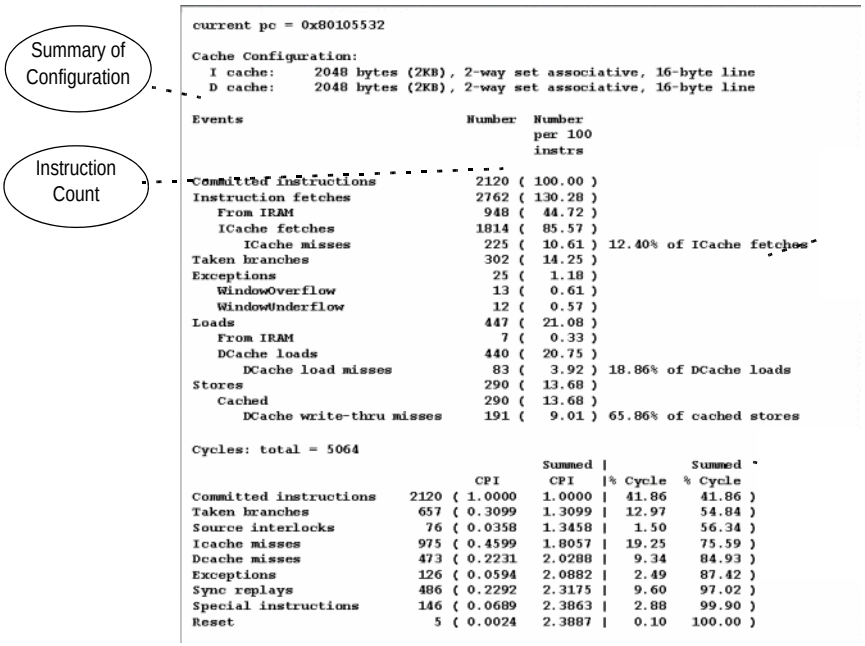
The Xtensa simulation environment includes the stand-alone Instruction Set Simulator (ISS) and two APIs for accessing the ISS in library form: the Xtensa Modeling Protocol (XTMP) and Xtensa SystemC (XTSC) package. This simulation environment enables fast and accurate simulation of system-on-chip designs incorporating one or more processor cores. Running at speeds that are orders of magnitude faster than RTL simulators, the Xtensa ISS is a powerful tool for software development and SOC design.

The ISS is also flexible: it can serve as a back end for debugging with Xtensa Xplorer or the command-line debugger (`xt-gdb`), generate profiling data for Xtensa Xplorer or the command-line profiler (`xt-gprof`), or operate as a system-simulation component in third-party co-simulation products.

The ISS has two major simulation modes: cycle-accurate simulation and fast functional simulation (TurboXim, a premium option).

5.6.1 Cycle-Accurate Simulation

In the cycle-accurate mode, the ISS directly simulates multiple instructions in the processor's pipeline, cycle by cycle. The simulator models most micro-architectural features of the Xtensa processor and maintains a very high degree of cycle accuracy. When requested, the ISS provides a performance summary, which provides a detailed accounting of processors cycles and events that contributed to them. Figure 5–10 shows an example output from the simulator.




```
current pc = 0x80105532

Cache Configuration:
  I cache:     2048 bytes (2KB), 2-way set associative, 16-byte line
  D cache:     2048 bytes (2KB), 2-way set associative, 16-byte line

Events                                Number  Number
                                              per 100
                                              instrs

Committed instructions                2120 ( 100.00 )
Instruction fetches                   2762 ( 130.28 )
   From IRAM                           948 (  44.72 )
   ICache fetches                     1814 (  85.57 )
      ICache misses                    225 (  10.61 ) 12.40% of ICache fetches
Taken branches                         302 (  14.25 )
Exceptions                              25 (   1.18 )
   WindowOverflow                       13 (   0.61 )
   WindowUnderflow                      12 (   0.57 )
Loads                                  447 (  21.08 )
   From IRAM                             7 (   0.33 )
   DCache loads                        440 (  20.75 )
      DCache load misses                83 (   3.92 ) 18.86% of DCache loads
Stores                                 290 (  13.68 )
   Cached                              290 (  13.68 )
      DCache write-thru misses         191 (   9.01 ) 65.86% of cached stores

Cycles: total = 5064
                                                 Summed |          Summed
                                       CPI         CPI  |% Cycle  % Cycle
Committed instructions      2120 ( 1.0000    1.0000 |  41.86    41.86 )
Taken branches               657 ( 0.3099    1.3099 |  12.97    54.84 )
Source interlocks             76 ( 0.0358    1.3458 |   1.50    56.34 )
Icache misses                975 ( 0.4599    1.8057 |  19.25    75.59 )
Dcache misses                473 ( 0.2231    2.0288 |   9.34    84.93 )
Exceptions                   126 ( 0.0594    2.0882 |   2.49    87.42 )
Sync replays                 486 ( 0.2292    2.3175 |   9.60    97.02 )
Special instructions         146 ( 0.0689    2.3863 |   2.88    99.90 )
Reset                          5 ( 0.0024    2.3887 |   0.10   100.00 )
```

Figure 5–10. Instruction Set Simulator Sample Output

The Xtensa ISS is a software application that models the behavior of the Xtensa instruction set. While the ISS directly models the Xtensa pipeline and is thus quite accurate, unlike its counterpart HDL processor hardware model, the ISS abstracts the DPU during its instruction execution to speed simulation. As a result, there may be small cycle-count differences between ISS reports and actual hardware under special conditions that can disturb the normal pipeline operation. Even in the cycle-accurate mode, the ISS runs much faster than hardware simulation because it need not model every signal transition for every gate and register in the complete processor design.

5.6.2 TurboXim: Fast Functional Simulation

The ISS' fast functional simulation mode, called TurboXim, is available as an extra-cost option. TurboXim does not model micro-architectural details of the Xtensa processor but it does perform architecturally correct simulation of all Xtensa instructions and exceptions. Fast functional simulation using TurboXim can be 40 to 80 times faster than the cycle-accurate simulation, which makes it ideally suited for high-level functional verification of application software.

In the TurboXim mode, cycles lost due to pipeline stalls and replays, memory delays, and branch penalties are ignored. Each instruction is assumed to take a single cycle to complete. The fast functional mode and the cycle-accurate mode can be combined in the same simulator invocation, which allows you to generate very accurate profiles for long-running programs at speeds much faster than the cycle-accurate simulation can provide by itself.

5.6.3 XTMP: Xtensa Modeling Protocol

The XTMP simulation environment allows system designers to create customized, multi-threaded simulators of their systems using a C-based API to the Xtensa ISS. Designers can instantiate multiple Xtensa cores and connect them to customer-designed models of system peripherals and interconnects. The Xtensa processor cores and peripherals accurately communicate using a cycle-accurate, split-transaction simulation model. XTMP can be used for simulating homogeneous or heterogeneous multi-core systems, as well as complex single-processor system architectures. This allows system designers to create, debug, profile, and verify their combined SOC and software architecture early in the design process.

In addition to the simulation library interface, XTMP includes a pre-built multi-core simulator (`xtmp-run`) for easier out-of-the-box modeling of simple multiprocessor systems with shared memory.

5.6.4 XTSC: Xtensa SystemC Package

The Xtensa SystemC package (XTSC) provides interfaces for both transaction-level and pin-level modeling of Xtensa processor cores in SystemC simulation environments. In addition, XTSC pin-level interfaces can be used for SystemC-Verilog co-simulation with standard RTL simulators.

For Xtensa outbound and inbound PIF, each configured local memory, TIE port, TIE queue, TIE lookup, and select system signals, XTSC allows designers to individually select whether to model an interface at the transaction level or at the pin level.

The XTSC package includes the following:

- The XTSC core library, which gives access to the Xtensa ISS in the form of a SystemC module (`sc_module`).
- The XTSC component library, which contains a set of configurable SOC example components in the form of SystemC modules. Components with transaction-level interfaces include memories, arbiters, routers, queues, lookups, etc. Components with SystemC pin-level interfaces are memories, queues, and lookups.

- The `xtsc-run` program, which can be use for two distinct purposes:
  - To quickly build and simulate arbitrary XTSC systems comprised of any number of modules from the XTSC core or component library, or
  - To generate a glue layer needed for co-simulation of an XTSC system with one or more existing Verilog modules (SystemC-on-top), or co-simulation of an existing Verilog system with one or more XTSC modules (Verilog-on-top).
- A set of example projects that illustrate Xtensa core and component library transaction-level and pin-level interfaces in a pure SystemC environment, and a set of example projects that illustrate co-simulation between XTSC and Verilog modules.

5.6.5 Simulation Speed

Table 5–10 compares the estimated speed of various simulation options you have when you design with the Xtensa processor. The simulation speed depends on the host computer and the complexity of your Xtensa processor instruction set. The numbers presented here are estimates based on simulating a typical range of Xtensa processor configurations on a 3GHz Intel Xeon 5160 processor running Linux.

Table 5–10. Instruction Set Simulator Performance Comparisons with Other Tools

Simulation Speed (MIPS or cycles/seconds)	Modeling Tool	Benefits
20 to 50 MIPS[1]	Stand-alone ISS in TurboXim mode	▪ Fast functional simulation for rapid application testing
1.5 to 40 MIPS[1,2]	XTMP or XTSC in TurboXim mode	▪ Fast functional simulation for rapid application testing at the system level
800K to 1,600K cycles/second	Standalone ISS in cycle-accurate mode	▪ Software verification ▪ Cycle accuracy
600K to 1,000K cycles/second[2]	XTMP in cycle-accurate mode	▪ Multi-core subsystem modeling ▪ Cycle accuracy
350K to 600K cycles/second[2]	XTSC in cycle-accurate mode	▪ Multi-core subsystem modeling ▪ SystemC interfaces ▪ Cycle accuracy
1K to 4K cycles/second	RTL Simulation	▪ Functional verification ▪ Pipeline-cycle accuracy ▪ High visibility and accuracy
10 to 100 cycles/second	Gate-level Simulation	▪ Timing verification ▪ Pipeline-cycle accuracy ▪ High visibility and accuracy

1. TurboXim mode simulation speed is an estimate for relatively long-running application programs (1 billion instructions or more).
2. Simulation speed is an estimate for a single Xtensa core in XTMP or XTSC.

5.7 Xtensa Debug Module Features

The Xtensa processor offers an optional Xtensa Debug Module that provides comprehensive debug support through a variety of features.

5.7.1 Functional Description of the Xtensa Debug Module

The Access Port provides external access to the internal, software-visible processor state through a narrow port connected directly to the processor core. The port uses the IEEE 1149.1 (JTAG) port or debug APB for communications with an external probe. Via the Access Port, it is possible to do all of the following:

1. Generate an interrupt to put the processor in the debug mode
2. Read any program-visible register or memory location
3. Modify any program-visible register or memory location
4. Modify the processor's PC to jump to a desired code location
5. Return to normal operating mode

5.7.2 Xtensa OCD Daemon

XOCD (the Xtensa OCD daemon), allows direct control over processor cores connected through the Access Port and does not require a stub running on the target. The target core can be stopped at any time to examine the state of the core, memory, and program. XOCD provides greater target visibility than monitor-based target debuggers and allows debugging of reset sequences, interrupts, and exceptions. XOCD implements a GDB-compatible stub interface and works with any debugger that supports the GDB remote protocol over a TCP/IP connection.

5.7.3 Multiple Processor On-Chip Debug

Having multiple processors in the same SoC (or SoC subsystem) introduces complexity in software debugging. A powerful tool for debugging multiple-processor designs is available for customers who have configured the *System Modeling and Multiple-Processor Support* option. This capability enables one Xtensa processor to selectively communicate a Break to other Xtensa processors in a multiple-processor system. Two interfaces are present to achieve this - BreakIn and BreakOut.

The BreakOut interface is used to communicate a break command to other processors when a processor has entered `OCDHaltMode`. The BreakIn interface is used to cause a processor to take a debug interrupt. As shown in Figure 5–11 below, these interfaces

can be used in concert to synchronously stop all cores in an entire MP system through a debug interrupt to a single core. By stopping the entire MP system in this manner, there is less perturbation of the system around the debug point of interest.

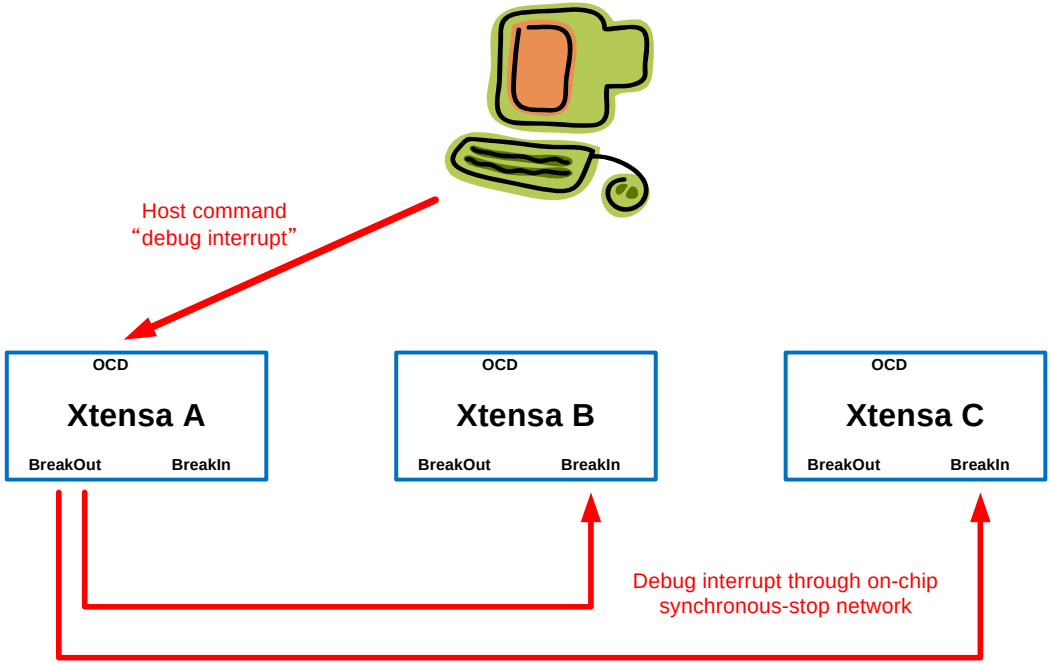


Figure 5–11. Synchronously stopping multiple cores

The DebugStall feature compliments BreakIn/BreakOut by providing a faster method to synchronously stop and resume a multiprocessor system for debug purposes. The debug state of the target core is communicated to other cores and causes these to be stalled. Another aspect of the DebugStall feature is that even when a core is stalled, debug operations are permitted to the core.

More detail on BreakIn/BreakOut and DebugStall is available in the "On-Chip Debug Implementation" chapter of the *Xtensa Debug Guide*.

5.7.4 OCDHaltOnReset

The Xtensa processor core provides an interface to force entry into `OCDHaltMode` when the processor comes out of reset. `OCDHaltOnReset` is an input signal to the Xtensa core that must be driven by external logic. In most cases, the signal will be driven low to allow normal operation when the Xtensa processor exits reset and starts fetching and executing instructions from memory. Alternatively, the signal can be driven high to force the processor to enter `OCDHaltMode` at reset. This mode is useful for debugging or for

downloading code through the processor's OCD port from a separate supervisor processor. The `OCDHaltOnReset` signal has no effect if it is changed during normal processor operation after the processor exits the reset state.

5.8 RTOS Support and the OSKit™ Targeting Environment

Linker support packages and an OSKit RTOS targeting environment are available for Xtensa processors. The linker support package includes linker scripts, exception handler examples, and C runtime examples for kernel development. Xtensa RTOS support consists of a generic HAL (a hardware-abstraction layer with a configured header and library files). Various operating systems—including Mentor Graphics' Nucleus, Express Logic's ThreadX, Micrium's μC/OS, and Linux—use this HAL to support various Xtensa processor configurations.

6. TIE for Software Developers

This chapter introduces the Cadence Tensilica Instruction Extension (TIE) language and its features.

6.1 Adapting the Processor to the Task

Processors have traditionally been extremely difficult to design and modify. As a result, most systems contain processors that were designed and verified once for general-purpose use, and then embedded into multiple applications over time. It then becomes the software developer's responsibility to tune the application code to run well on this processor. Optimizing code to run efficiently on a general-purpose processor is a labor intensive process, with the developer spending countless hours trying out various techniques to make the code run faster. Would it not be better instead to have a processor whose instruction set can be customized for a specific application or class of applications?

The Tensilica Instruction Extension (TIE) language provides the software developer with a concise way of extending the Xtensa processor's architecture and instruction set. In most cases, the developer simply describes the semantics of a set of desired custom instructions. From this description, the Xtensa Processor Generator and the TIE compiler automatically generate the hardware that decodes the new instructions and integrates their execution semantics into the Xtensa pipeline. In addition, the Xtensa Processor Generator and the TIE compiler generate a complete set of software tools including a compiler, assembler, simulator, and debugger that are each customized for the generated processor core.

6.2 TIE Development Flow

Instruction-set extensions are typically used to accelerate both compute- and I/O-intensive functions, or "hot spots," in a particular piece of application code. Because a hot spot is a region of code that is exercised heavily by the program, the efficiency of its implementation can have a significant effect on the efficiency of the whole program. It is possible to increase the performance of an application by a factor of 2, 5, and sometimes by an order of magnitude or more by adding a few carefully selected instructions to the processor.

TIE can be used in two ways: through the Xtensa Processor Generator and locally, through the TIE compiler. Local use of the TIE compiler allows developers to rapidly experiment with new instructions to quickly tune and optimize application code using ISA

extensions. Locally developed TIE then serves as an input to the Xtensa Processor Generator, which generates a processor with the ISA extensions and verifies the complete configuration.

Using the profiling tools available in the Xtensa Xplorer environment or using the `xt-gprof` utility together with the instruction set simulator (ISS), you can easily find an application's performance-critical code sections. Performance tuning is often an iterative process as illustrated in Figure 6–12. You can look at a region of code and experiment with different TIE instructions. After you have identified a critical code segment that can benefit from the addition of a TIE instruction, you describe the instruction in TIE and invoke the TIE compiler.

The TIE compiler generates a set of files that are used to configure all the software development tools so that they recognize your TIE extensions. The TIE compiler also provides a preliminary estimate of the hardware size of your new instruction to help you evaluate your new instruction. You now modify your application code as described in the next section to take advantage of the new instruction and simulate using the ISS to verify correct operation. You can look at the generated assembly code and rerun the profiler to evaluate performance. At this point you can decide if the performance improvement warrants the additional hardware cost and whether you should keep, remove, or modify the instruction. The turnaround time for this process is very short, because the software tools are automatically reconfigured for your new TIE.

Typically, a TIE instruction is first implemented with a direct, simple ISA description (instructions plus registers) using the TIE *operation* construct. After you are satisfied that the instructions you create are correct and that your performance goals are being met, you can optimize the TIE descriptions for a more efficient hardware implementation and to share hardware with other instructions using the TIE *semantic* construct. The hardware implementation can be verified against the simple ISA implementation using formal techniques, if desired.

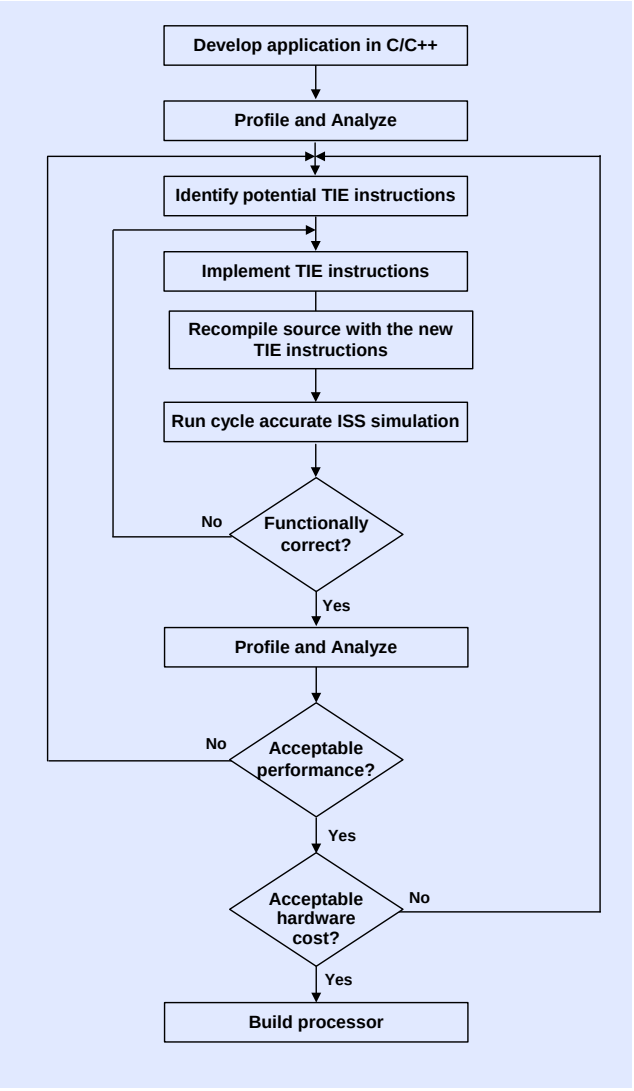


Figure 6–12. Typical TIE Development Cycle

6.3 Improving Application Performance Using TIE

TIE allows you to improve the performance of your application by enabling you to create one TIE instruction that does the work of multiple instructions of a general-purpose processor. There are several different techniques that you can use to combine multiple operations into one.

Two common techniques available through TIE are fusion and SIMD/vector transformation. To illustrate these techniques, consider a simple example where profiling indicates that most of a program's execution time is spent in the following loop, computing the average of two arrays.

```
unsigned short *a, *b, *c;
...
for (i=0; i<n; i++)
     c[i] = (a[i] + b[i]) >> 1;
```

6.3.1 Fusion

In the above piece of C code, you can see that in every iteration of the loop, two *short* data items are being added together and the result is being shifted right by one bit. Two Xtensa core instructions are required for the computation, not counting the instructions required for loading and storing the data. These two operations can be fused into a single TIE instruction, which can be fully described as follows.

```
operation AVERAGE{out AR res, in AR input0, in AR input1} {} {
     wire [16:0] tmp = input0[15:0] + input1[15:0];
     assign res = tmp[16:1];
}
```

The instruction takes two input values (`input0` and `input1`) from the Xtensa core AR register file to compute the output value (`res`). The result is then put into the AR register file. The semantics of the instruction, an add feeding a shift, are described using standard Verilog-like syntax. To use the instruction in C or C++, simply modify the original example code as follows.

```
#include <xtensa/tie/average.h>
unsigned short *a, *b, *c;
...
for (i=0; i<n; i++)
     c[i] = AVERAGE(a[i], b[i]);
```

The AVERAGE instruction can also be used directly in assembly code. The entire software toolchain recognizes AVERAGE as a valid instruction for this processor. The TIE compiler implements this instruction in a manner consistent with the Xtensa processor's pipeline. Thus, the TIE compiler automates the task of implementing instruction extensions, both in hardware as well as in software. This allows the software developer to focus on the semantics of their instruction extensions, without worrying about the implementation details.

Fused instructions are not necessarily as expensive in hardware as the sum of their constituent parts. Often they are significantly cheaper because they operate on restricted data sets. In this example, the addition inside AVERAGE is just a 16-bit addition, while

the Xtensa instruction set implements 32-bit additions. Further, because the data is shifted by a fixed amount (right shift by 1), the shift operation is essentially free of hardware cost because the fixed 1-bit shift operation can be performed by simply extracting the appropriate bits from the result of the addition.

The AVERAGE instruction requires very few gates and easily executes in one cycle. However, TIE instructions need not execute in a single cycle. The TIE *schedule* construct allows you to create instructions with computations that span multiple clock cycles. Such instructions can be fully pipelined (multiple instances of the instruction can be issued back to back) or they may be iterative. Fully pipelined instructions may achieve higher performance because they can be issued back-to-back, but they may also require extra implementation hardware. An iterative instruction spanning multiple clock cycles uses the same set of hardware, over two or more clock cycles. This design approach saves hardware but if the application attempts to issue back-to-back iterative instructions during adjacent cycles, the processor will stall until the first dispatch of the instruction completes its computation (stops using the shared resource).

6.3.2 SIMD/Vector Transformation

In the example of fusion shown above, consecutive add and shift operations are combined into one instruction. Other types of instruction combinations can also improve performance. For example, in every iteration of the loop, the example code performs the same computation on new data. You can use SIMD (single-instruction, multiple-data) instructions (also called vector instructions) to compute multiple iterations of a computation using one instruction. You can also combine fusion and SIMD techniques. Consider, for example, the case where you would like to compute four averages in one instruction. This can be done using the following TIE description:

```
regfile VEC 64 8 v

operation VAVERAGE{out VEC res, in VEC input0, in VEC input1} {} {
     wire [67:0] tmp = {input0[63:48] + input1[63:48],
                        input0[47:32] + input1[47:32],
                        input0[31:16] + input1[31:16],
                        input0[15:0] + input1{15:0]};
     assign res = {tmp[67:52], tmp[50:35], tmp[33:18], tmp[16:1]};
}
```

Note that computing four 16-bit averages requires that each data vector be 64 bits (4x 16 bits) and the Xtensa AR register file is only 32 bits. Therefore, we create a new register file, called VEC, with eight 64-bit registers to hold the 64-bit data vectors for the new SIMD instruction. This new instruction, VAVERAGE, takes two operands from the VEC register file, computes four averages with the 1-bit shift, and saves the 64-bit vector result in the VEC register file. To use the instruction in C/C++, simply modify the original example as follows.

```
#include <xtensa/tie/average.h>
VEC *a, *b, *c;
...
for (i=0; i<n; i+=4) {
     c[i] = VAVERAGE(a[i], b[i]);
}
```

The automatically generated Xtensa processor compiler recognizes a new 64-bit C/C++ data type VEC, corresponding to the new register file. In addition to VAVERAGE, the TIE compiler automatically creates new load and store instructions to move 64-bit vectors between the VEC register file and memory. The C compiler uses these instructions to load and store the 64-bit arrays of type VEC.

Note that the Xtensa processor's local data-memory ports (data RAM, data ROM, and data cache) must be configured with a width of at least 64 bits to move 64-bit data between memory and the new 64-bit VEC register file. This requirement does not apply to the processor's PIF bus.

Compared to the fused instruction AVERAGE, the vector-fused instruction VAVERAGE requires significantly more hardware because it performs four 16-bit additions in parallel (in addition to the four 1-bit shifts, which do not require any additional gates). However, the performance improvement from combining vectorization and fusion is significantly larger than the performance improvement from fusion alone.

For instructions with semantics that naturally map into standard C operators, it is possible to overload C operators to automatically apply to TIE data types, thereby avoiding the need for intrinsics. For example, if a vector add instruction, VADD, is created in addition to VAVERAGE, and the following is added to the TIE description:

```
operator "+" VADD
```

Then the following C code can be used to perform vector addition.

```
VEC *a, *b, *c;
...
for (i=0; i<n; i+=4) {
   c[i] = a[i] + b[i];
}
```

6.3.3 FLIX

A third technique to perform multiple operations in a single instruction involves the Flexible Length Instruction Xtensions (FLIX) capabilities of the Xtensa LX7 processor. Where instruction fusion combines dependent operations in a single instruction and SIMD/vector transformation replicates the same operation multiple times in one instruction, FLIX combines *unrelated* operations in one instruction word.

6.3.4 Combining Independent Operations with TIE

Fusion combines dependent operations in a single instruction. SIMD/vector transformation replicates the same operation multiple times in one instruction. A SIMD/vector operation has no more operands than the original scalar instruction. Because intermediate results and operands in fused instructions never go into or out of the register file, a fused instruction typically has a similar number of operands as the original instructions being fused. In contrast, when combining unrelated operations into one instruction word, the number of operands grows in proportion to the number of operations being combined. If many or all of these operands reside in one register file, this operand growth caused by the increase in the number of parallel operations may affect the number of read and write ports required on that register file.

Xtensa LX7 core instructions are either 16- or 24-bits wide. It can be difficult to combine two or more unrelated operations into one instruction and encode all the required operands in just 24 bits. Therefore, Xtensa LX7 processor TIE instructions can be 24 bits in size up to 128 bits in 8-bit increments. Instructions of different sizes can be freely intermixed. There is no “mode bit” for instruction width. Each instruction’s size is encoded into the instruction word itself. The Xtensa LX7 processor identifies, decodes, and executes any mix of instruction sizes from the incoming instruction stream. The wide instructions (meaning instructions between 32 bits and 128 bits in size) can be divided into slots, with independent operations placed in all or some of the slots. FLIX operation slots need not be equally sized, which is why this feature is called Flexible Length Instruction Xtensions (FLIX). Any combination of the operations allowed in each slot can occupy a single FLIX instruction word.

Consider again the AVERAGE example. On a base Xtensa LX7 processor, the inner loop contains the ADD and SRAI instruction to do the actual computation, and two L16UI load instructions and one S16I store instruction as well as three ADDI instruction to update the address pointers used by the loads and stores. To accelerate this code, you could create a 64-bit FLIX instruction format with one slot for the load and store instructions, one slot for the computation instructions, and one slot for address-update instructions. This can be done using the following TIE description:

```
format flix210 64 {slot0, slot1, slot2}

slot_opcodes slot0 {L16UI, S16I}
slot_opcodes slot1 {ADDI}
slot_opcodes slot2 {ADD, SRAI, ADDI}
```

The first declaration creates a 64-bit instruction and defines an instruction format with three opcode slots. The last three lines of code list the instructions available in each opcode slot defined for this FLIX configuration. Note that all the instructions specified are predefined, core processor instructions, so their definition need not be provided in the TIE code because the TIE compiler already knows about all core Xtensa LX7 instructions.

For this example, no changes are needed to the C/C++ program. The generated C/C++ compiler will compile the original source code, taking advantage of FLIX automatically. For example, the generated assembly code for this processor implementation would look like this:

```
loop:
{ l16ui a11, a8, 0;      addi a8, a8, 2; add  a10, a10, a11 }
{ l16ui a10, a9, 0;      addi a9, a9, 2; srai   a6, a10, 1}
{ s16i a6, a4, 0;        addi a4, a4, 2; nop      }
```

Thus, a computation that would have taken eight cycles per iteration on a base Xtensa LX7 processor now takes just three cycles per iteration. Only five lines of TIE code were required to specify a complex FLIX configuration format with three instruction slots using existing Xtensa LX7 opcodes, as shown in the following table.

Table 6–11. Performance Tuning with Fusion, SIMD, and FLIX Instruction Extensions

	Number of Cycles Per Iteration of “AVERAGE” Function	Number of Iterations of "AVERAGE" Function
Base Processor	8	n
Processor with Fused Instructions	7	n
Processor with SIMD Instructions	7	n/4
Processor with FLIX Instructions	3	n

This simple example illustrates how to place Xtensa LX7 core instructions in multiple instruction slots. It is equally easy to create a FLIX configuration with slots that contain a mix of Xtensa core instructions and designer-defined TIE instructions.

FLIX has several advantages compared to the other two techniques. No changes to the C/C++ source are required when you use FLIX to replicate Xtensa core instructions in multiple instruction slots. A particular FLIX implementation is more likely to apply to a wider range of C/C++ programs than a TIE implementation using fusion or SIMD/vector transformation.

However, the other techniques also have advantages. Fusion often adds just a small amount of hardware. SIMD instructions tend to give larger performance improvements. Which technique you use depends on your goals, budget, and application. Of course, all the techniques can be combined. Because our processor generator is very fast, it’s easy to experiment with different approaches to find the fastest, most efficient Xtensa LX7 processor architecture for a target application.

Note that even without FLIX and its wide 32- to 128-bit instructions, it is possible to combine independent operations into a single instruction. It is *not* possible to fit two operations, each referring to three register operands, into a 24-bit instruction because of the number of bits needed to encode the register-file-address operands. However, TIE al-

lows you to create STATE registers that are implicitly addressed so they do not take any instruction-word bits for operand-source or result-destination encoding. Using the generated assembly from the previous FLIX example as a template, you could create three separate TIE instructions operating on six pieces of state, labeled s_va through s_pc. The first instruction, for example, increments s_pa by four, sets s_vc to the sum of s_vb and s_va, and loads from the contents of s_pa into s_va. Because each instruction always reads or writes a specific set of states rather than one of many register-file entries, only a few bits are required to encode the instruction.

A full TIE description of this example follows:

```
        state s_va 16 add_read_write
        state s_pa 32 add_read_write
        state s_vb 16 add_read_write
        state s_pb 32 add_read_write
        state s_vc 16 add_read_write
        state s_pc 32 add_read_write

    // Load A element, Increment A pointer, Add previous A and B elements,
    place in C element
    operation LDA_INCPA_ADD{} {inout s_pa, out s_vc, in s_vb,inout s_va,
              out VAddr, in MemDataIn32} {
        assign VAddr = s_pa;
        assign s_pa = s_pa + 32'd4;
        assign s_vc = s_vb + s_va;
        assign s_va = MemDataIn32;
    }

    // Load A element, Increment A pointer, Shift Right C element
    operation LDB_INCPB_SRC{}{inout s_pb, inout s_vc, out s_vb,
              out VAddr, in MemDataIn32} {
        assign VAddr = s_pb;
        assign s_pb = s_pb + 32'd4;
        assign s_vc = {1'd0, s_vc[15:1]};
        assign s_vb = MemDataIn32;
    }

    // Store C element, Increment C pointer
        operation STC_INCPC{}{inout s_pc, in s_vc,
              out VAddr, out MemDataOut32} {
        assign VAddr = s_pc;
        assign s_pc = s_pc + 32'd4;
        assign MemDataOut32 = s_vc;
    }
```

This example illustrates custom load and store instructions. By setting the VAddr interface, the core load/store unit is instructed to perform a load or store operation from or to the programmer-defined address. The data from a load operation is returned to the TIE

execution unit via the MemDataIn32 interface. Data to be stored is assigned via the TIE execution unit to the MemDataOut32 interface. When a TIE state is declared with the attribute *add_read_write*, the TIE compiler automatically creates instructions to move the state to and from the general-purpose AR register file. The read instruction has the mnemonic RUR.<state name> and the write instruction has the mnemonic WUR.<state name>.

To use these instructions in C/C++, modify the original example as follows.

```
#include <xtensa/tie/average.h>
unsigned short *a, *b, *c;
...
{     short i=0;
      WUR_s_pa(a);
      WUR_s_pb(b);
      WUR_s_pc(c);
      WUR_s_va(i);
      WUR_s_vb(i);

      for (i=0; i<n; i++) {
            LDA_INCPA_ADD();
            LDB_INCPB_SRC();
            STC_INCPC();
      }
}
```

The first three instructions initialize the state registers to the addresses of the arrays. Inside the loop, the three instructions cycle through the computation. Note that none of these instructions take any arguments because they operate only on TIE state, and they allocate no bits to specify a register-file address or immediate operands.

This style of combining independent operations can possibly provide performance gains comparable to gains achieved by using the Cadence FLIX technology, but with more modest hardware costs. However, this type of TIE tends to be less flexible and more difficult to program than FLIX-based extensions. If a different application needs to perform adds, shifts, and loads in parallel, but data for these operations comes from or results go to different states, the instructions created by the original TIE would not be general enough to meet these requirements. The FLIX approach is more appropriate in this situation.

6.4 TIE Queue Interfaces, Lookups, and Ports

The TIE features described so far communicate to the world outside of the processor only through the processor's memory and PIF bus interfaces. In many embedded applications, the processor must communicate with other devices in the system in addition to memories. The processor might also need access to local storage with much greater

bandwidth than is provided by the processor's existing memory interfaces. The system design may require direct processor-to-processor communications without incurring the overhead of shared memory. The TIE language provides three convenient and flexible ways for the processor to communicate with one or more external devices: TIE queue interfaces, lookups, and ports.

6.4.1 TIE Queue Interfaces

A FIFO queue is commonly used to communicate between different devices in a System-On-Chip (SOC) design. The TIE language provides constructs that allow the Xtensa LX7 processor to directly interface with input or output queues. Designer-defined instructions can "pop" an entry from an input queue and use the incoming data for a computation or for flow control. Similarly, the results of a computation (or alternately, a command packet for another device) can be "pushed" onto an output queue.

After a queue interface is defined in TIE, the TIE compiler automatically generates all the interface signals for the additional interface port needed to connect to the external FIFO. The logic to stall the processor when it attempts to read from an empty input queue or attempts to write to a full output queue is also automatically generated. The language semantics also allow conditional queue access. A run-time condition setting can determine if an instruction that writes to an output queue actually performs the write or not. Similarly a run-time condition can be used to determine if an instruction that reads from an input queue actually consumes the data or not.

6.4.2 TIE Lookups

A TIE "lookup" is used to send a request or an address to an external device and to receive the response or data from the external device one or more cycles later. This interface can be used, for example, to interface to an external lookup table stored in a ROM— hence the name "lookup." However the TIE lookup interface has additional features that makes it an ideal choice to connect to several other types of non-memory hardware blocks external to the processor. For example, an external device connected to a processor via a TIE lookup port can stall the processor if it is not ready to accept the lookup request.

6.4.3 TIE Ports

The TIE language for the Xtensa LX7 processor also supports an "import-wire" construct that the processor uses to directly sample the value of an external input signal and an "export-state" construct that the processor uses to drive the value of any TIE state onto externally accessible output signals. These input and output signals are collectively called TIE ports.

Note: For details on these features, see Chapter 24.10.1 “TIE Ports” . System design aspects of TIE ports, queues, and lookups are also described in Section 24.11.

6.4.4 TIE Port and Queue Wizard

Xtensa Xplorer provides a simple wizard for creating ports and queues without the need to write any TIE.

After specifying the width and direction of each port or queue, the wizard generates a TIE file that gets included in the processor configuration. A software project with test inputs is also created which can be run to exercise the created interfaces.

As the wizard output is a TIE file, this can be edited or deleted from the configuration and new interfaces can be added by re-running the wizard as many times as needed.

Refer to the following Xtensa Xplorer Help topics for detailed information: TIE Port and Queue Extensions, Using the TIE Port Queue Wizard, and TIE Port Queue Wizard.

7. Performance Programming Tips for the Software Developer

Through TIE, Cadence gives you the ability to tailor your processor to your application, potentially giving you performance levels not possible with traditional processors. However, even without TIE, there is much you can do to help maximize your application's performance, and Cadence provides several tools to help. Furthermore, cleaning and tuning your code can often make it even more amenable to TIE acceleration, particularly when using TIE through automated tools.

7.1 Diagnosing Performance Problems

Without tools, it can be difficult to know where your performance problems are or what you can do to solve them. For example, a floating-point multiplication using software emulation might take approximately 100 cycles. If 1% of the operations in your application are floating-point multiplies, the multiplication routine represents significant computational overhead. If 0.01% of the operations are floating-point multiplies, the overhead might be negligible.

There are two classes of tools that can help you diagnose software-performance problems.

7.1.1 Profiling

Profiling (available in simulation or on real hardware through Xplorer as well as through `xt-gprof` shows you where your application code is spending the most time, either at the function level or at the source-line or assembly-line level. You should, of course, concentrate your efforts on regions that consume a significant portion of the time.

7.1.2 Simulation

When using simulation, the simulator is able to break down the time spent executing an application event-by-event. For example, the simulator can tell you how much time is spent globally servicing cache misses or accessing uncached memory. This information helps you know whether to concentrate your efforts on generated code sequences or on your memory subsystem. Profiling and simulation data can also be combined. You can profile code based on cache misses, for example, to see which portions of a program cause the largest number of cache misses.

7.2 Types of Performance Problems

When you know where to start, you can begin to tune your code. There are several aspects to performance including algorithmic, configuration, memory system, microarchitectural, and compiler code quality. We will discuss each in turn.

7.3 Choosing Algorithms

Choice of algorithm has a first-order effect on performance. A radix-4 FFT fundamentally has fewer multiplies than a radix-2 FFT. A heap sort fundamentally requires fewer compares than an insertion sort. Algorithm choice is mostly independent of processor architecture, and there are few tools to help select among alternative algorithms. However, there is some connection between algorithm and processor or system architecture.

For example, a radix-4 FFT requires fewer multiplies than a radix-2, and a radix-8 FFT requires fewer multiplies than either of the other two types. However, as you increase the FFT radix, you increase the operation complexity and the number of registers required. At sufficiently high radix, performance becomes dominated by overhead and data shuffling. A typical Xtensa processor configuration tends to handle radix-4 better than radix-8. Be mindful of your architecture when choosing algorithms and, for the Xtensa processor, be mindful of your algorithm when tuning the architecture.

7.4 Configuration

The choice of processor configuration can greatly affect performance. As previously described, the use of TIE can increase performance by many factors, but configuration parameters and choice of memory subsystem can also make a substantial difference.

There are two basic techniques that can be used to improve your choice of core processor configuration. Using the first technique, profile your application and take a closer look at the hotspots. If profiling shows, for example, that much time is spent in integer multiplication emulation, adding a multiplier to your configuration will probably help. Using the second technique, build multiple configurations and see how performance of your application varies across configurations. Xtensa Xplorer allows you to manage multiple configurations and graphically compare performance of your application using different configurations.

In Figure 7–13, we use Xtensa Xplorer to compare performance of an application performing floating-point multiplies using two configurations; one with the Xtensa floating-point unit (FPU) and one without. The figure shows two sets of bars. The left-hand set of three bars shows software cycle counts for a processor configuration with an FPU. The right-hand set of bars shows a processor configuration without an FPU. Within each set

of three bars, the first bar shows the total cycle count for the software routine, the second bar shows the branch delays, and the third bar shows the interlock cycles. As shown in Figure 7–13, the floating-point unit greatly improves performance.

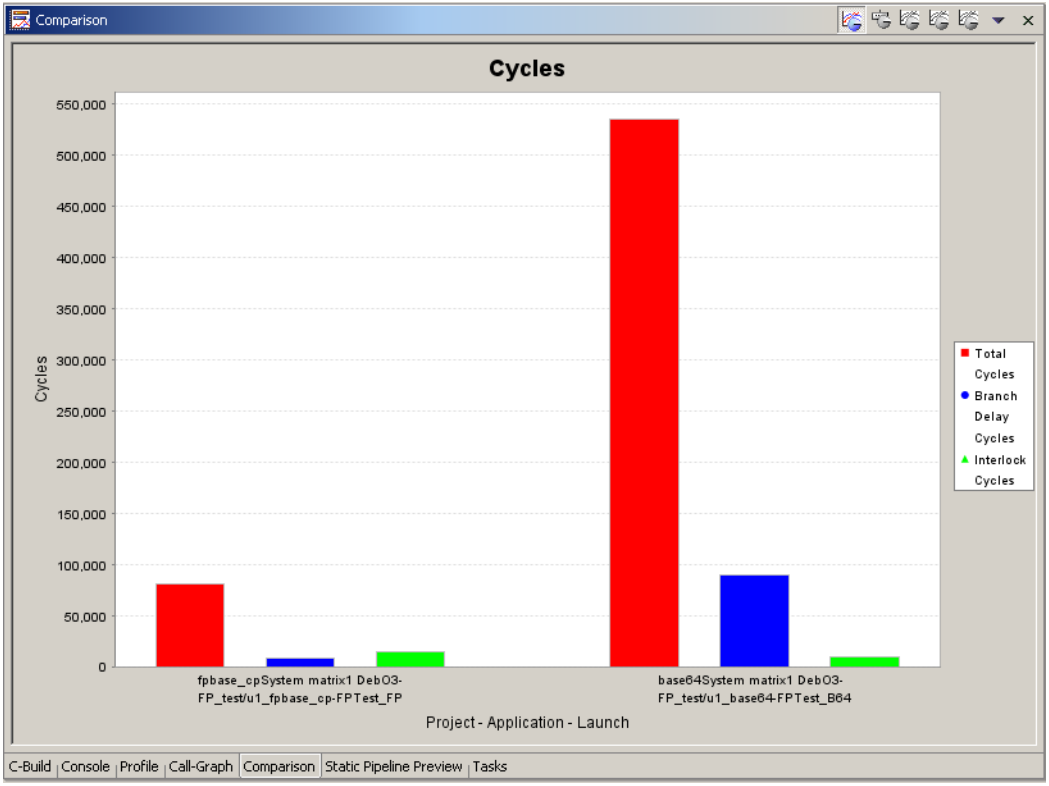


Figure 7–13. Adding a Floating-Point Unit to a Configuration

7.5 Memory System

In modern embedded systems, how fast you get data into and out of the processor can often be the most critical aspect of performance. Xtensa processors give you a lot of freedom in designing external interfaces. Communication can be through local memory ports, the Xtensa Local Memory Interface (XLMI), the PIF, caches, TIE ports, TIE queue interfaces or TIE lookups. Each of these interface types can be tailored for the application. You can, for example, choose the width of the PIF, the size of caches (if any), and the number of memory units.

Cadence provides several tools to measure and analyze memory-system performance in simulation. The standalone ISS simulator accurately models caches and the interactions between the memory system and the pipeline for a fixed-latency/fixed-bandwidth

memory system. XTSC and XTMP allow you to accurately model more complicated memory systems including DMA, TIE queue interfaces, ports and lookup, and custom devices and interconnects.

The profiler, either through xt-gprof or through Xtensa Xplorer, allows you to profile your application code based on cache misses, allowing you to see which regions of your code suffer the most cache misses or spend the most time handling cache misses.

As with other aspects of configuration, Xtensa Xplorer allows you to easily compare performance of an application using multiple memory systems. In addition, the Cache Explorer feature of Xtensa Xplorer, shown in Figure 7–14, allows you to graphically select a range of cache parameters and have the system automatically simulate and compare performance for all selected configurations.

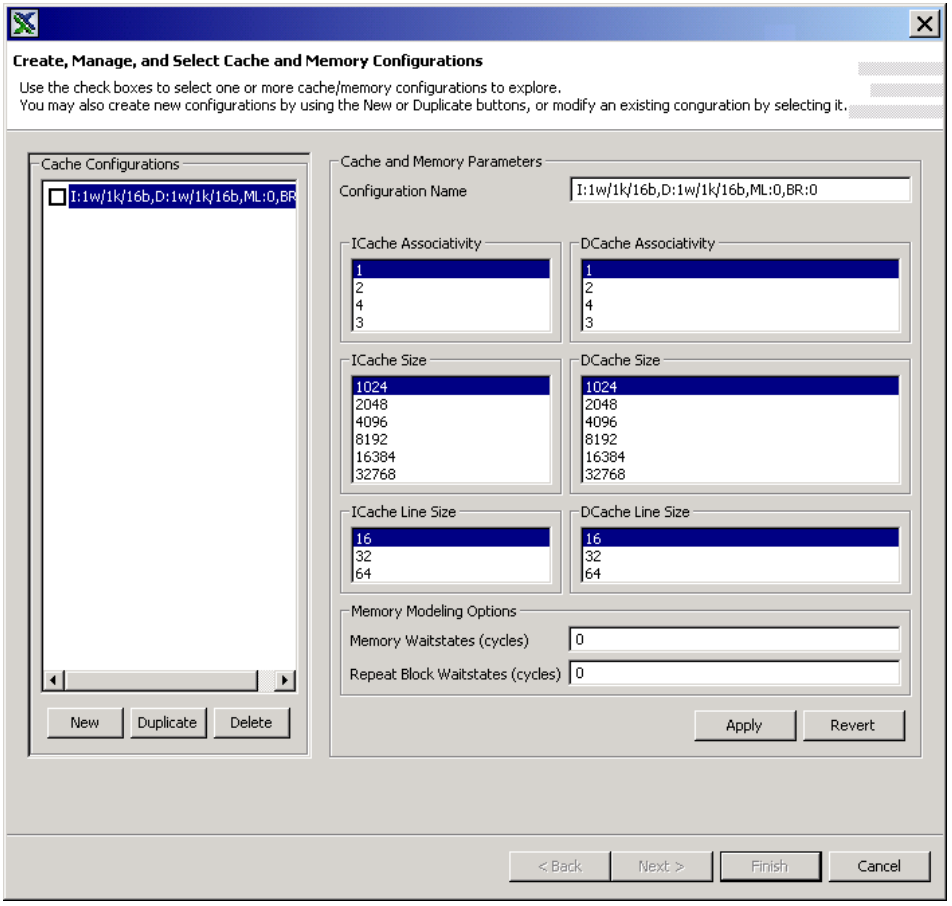


Figure 7–14. Cache Explorer

When tuning your memory subsystem, it is important to model a system and application similar to the one you actually will use. For example, on a system with local memories, DMA might significantly improve performance for some applications. Simulating such applications without modeling DMA might lead one to incorrect conclusions about the required cache parameters.

By default, the stand-alone ISS does not simulate the memory subsystem; it assumes zero delay cycles on all loads and stores. This policy allows you to tune the other aspects of performance independently of your memory subsystem. Particularly with caches, small changes that improve microarchitectural performance might randomly hurt memory-system performance as they change data or instruction layout. Nonetheless, all these small changes in aggregate will probably improve performance. Simulations with an ideal memory system allow you to ignore these effects until you are ready to tune your memory system. After architectural tuning of the processor is finished, you can add a memory model to the stand-alone ISS simulation, or use the XTSC or XTMP simulation environment to add non-zero memory-delay characteristics to the simulation.

7.6 Microarchitectural Description

At full speed, the Xtensa processor's pipeline can execute one instruction every cycle. However, various situations can stall the pipeline. The most common reasons for stalling the processor are memory delays, branch delays, and pipeline interlocks. The ISS summarizes the number of cycles stalled for different reasons. An example output is shown in Figure 7–15.

```
Cycles: total = 17893893

                                          Summed |           Summed
                                  CPI      CPI   |% Cycle  % Cycle
Committed instructions        15313272 ( 1.0000   1.0000 |  85.58    85.58 )
Taken branches                 1948514 ( 0.1272   1.1272 |  10.89    96.47 )
Source interlocks               106274 ( 0.0069   1.1342 |   0.59    97.06 )
Icache misses                   480983 ( 0.0314   1.1656 |   2.69    99.75 )
Dcache misses                     5246 ( 0.0003   1.1659 |   0.03    99.78 )
Exceptions                         835 ( 0.0001   1.1660 |   0.00    99.78 )
Uncached ifetches                 7493 ( 0.0005   1.1665 |   0.04    99.83 )
Uncached loads                      31 ( 0.0000   1.1665 |   0.00    99.83 )
```

Figure 7–15. Example Pipeline Delays

In addition, the profiler allows you to create profiles based on events, such as branch penalties. This feature allows you, for example, to identify functions, lines, or instructions in your program that suffer the most from branch penalties.

Because the processor's pipeline delays are small, pipeline interlocks often do not seriously degrade performance. Interlocks that do exist are often hard to avoid. As in the example in Figure 7–16, branch penalties are often larger. Except for zero-overhead loops, every aligned branch taken in the 5-stage version of the processor's pipeline suffers a 2-cycle branch penalty. One additional penalty cycle is required for unaligned branch targets (the instruction at the branch target address is not aligned so that the entire instruction fits in a single aligned fetch width (32-, 64-, or 128-bits depending on the fetch width option). A 7-stage version of the Xtensa LX7 pipeline adds one additional branch-penalty cycle.

Note: If the branch is not taken, there are no branch-penalty cycles.

The Xtensa C/C++ compiler (XCC) reduces branch penalties in two different ways. First, it inlines functions into their caller to avoid call and return branch delays. Note that when compiling with only the -O2 or -O3 compiler flags, the compiler can only inline functions when both the caller and the callee are in the same compilation unit or file. If you compile with the `-ipa` (interprocedural analysis) compiler flag, the compiler can inline any direct call regardless of file boundaries. Second, the compiler tries to lay out code so that the most frequent code traces become straight lines. Consider the example in Figure 7–16.


```
Statement 1;              Statement 1              Statement 1
if (cond) {               cond branch to L         cond branch to L
  Statement 2;            Statement 2              L3: Statement 3
}                         L: Statement 3           ...
Statement 3;                                       L: Statement 2
                                                   jump to L3:
```

Figure 7–16. Branch Directions

The compiler can choose to generate code that conditionally jumps around *Statement 2,* as shown in the center example. Alternatively, the compiler can generate code that jumps to someplace else that contains *Statement 2* followed by a jump back to *Statement 3* as shown in the rightmost example. The code in the center example will suffer a taken branch penalty whenever *cond* is not true but no penalty whenever *cond* is true. The code in the rightmost example will suffer two taken branch penalties whenever *cond* is true but none whenever *cond* is not true. The best choice depends on how often *cond* is true during program execution.

By default, the compiler guesses branch frequencies. You can help the compiler in two ways. First, the compiler supports a pragma, *#pragma frequency_hint,* with values of *FREQUENT* or *NEVER* that tell the compiler when a branch is very likely or very unlikely. Second, you can use the feedback feature of the compiler to develop run-time statistics about all code branches in a program.

7.7 Compiled Code Quality

Looking at your code, you often have expectations about the code that the compiler will generate. When tuning code, it is often helpful to take a close look at generated assembly code to see if it is as tight as you expect. There are many reasons why code quality might not be as good as expected. We will discuss some of the more common ones.

7.7.1 Choosing Compiler Flags

The compiler has many options to control the amount and types of optimizations it performs. Choosing the right options can significantly increase performance.

By default, the compiler compiles a file with *-O0* (no optimization). The compiler makes no effort to optimize the code either for space or performance. The main goals for unoptimized code generation are fast compilation and ease of debugging.

The lowest-level compiler optimization flag is *-O2*. The *-O3* compiler flag enables additional optimizations that are mostly (but not exclusively) useful for DSP-like code. Orthogonal to the use of *-O2* or *-O3*, the *-Os* tells the compiler to optimize for space rather than speed. The flag *-Os* can be used with either *-O2* or *-O3*; if neither is given, the compiler assumes *-O2*.

Xtensa DSP coprocessor engines (for example, ConnX D2, ConnX Vectra LX, HiFi 2, etc.) have the potential for speeding up standard C applications out of the box by using their DSP instructions and register files. The compiler, however, will not automatically infer the use of these instructions by default. This is because using these instructions might not be safe inside of interrupt handlers that do not save and restore the coprocessor state. Compile application code with the `-mcoproc` flag to tell the compiler that it is safe to use the DSP resources.

By default, the compiler optimizes one file at a time. The compiler is also capable of performing interprocedural analysis, which optimizes the entire application program across multiple source files. To use interprocedural analysis, add the `-ipa` flag to both the compiler and linker command lines. Behind the scenes, the compiler will delay optimization until the link step. The `-ipa` option can be used with or without *-Os*. The *-Os* works very well together with `-ipa` because interprocedural analysis allows the compiler to delete functions that are never used.

The compiler can make use of feedback data during the compilation process through use of the compiler flags *-fb_create* and *-fb_opt*. As mentioned in Section 7.6, the use of feedback enables the compiler to mitigate branch delays. In addition, the use of feedback allows the compiler to inline just the more frequently called functions and to place register spills in infrequent regions of code. The use of feedback allows the compiler to optimize for speed in the important portions of your application while optimizing for space everywhere else.

7.7.2 Aliasing

Consider the code in Figure 7–17. You might assume that the compiler will generate code that loads **a* into a register before the loop and contains an inner loop that loads *b[i]* into a register and adds it into the register containing **a* in every iteration.

```
void foo(int *a, int *b)
{
           int i;
           for (i=0; i<100; i++) {
                *a += b[i];
           }
}
```

Figure 7–17. Aliased Parameters

In fact, you will find that the compiler generates code that stores **a* into memory on every iteration. The reason is that *a* and *b* might be aliased; **a* might be one element in the *b* array. While it is very unlikely in this example that the variables will be aliased, the compiler cannot be sure.

There are several techniques to help the compiler optimize better in the presence of aliasing. You can compile using *-ipa*, you can use globals instead of parameters, you can compile with special flags, or you can annotate a variable declaration with the *__restrict* attribute.

7.7.3 Short Data Types

Most Xtensa instructions operate on 32-bit data. There are no 16-bit addition instructions. The system emulates 16-bit addition using 32-bit arithmetic and this sometimes forces the system to extend the sign of a 16-bit result to the upper bits of a register. Consider the example in Figure 7–18. You might expect that the routine would use a single add instruction to add *a* and *b* and place the result in the return register. However, C semantics say that if the result of *a+b* overflows 16 bits, the compiler must set *c* to the sign-extended value of the result's lower 16 bits, leading to the code in Figure 7–19.

```
int foo(short a, short b)
{
      short c;
      c = a+b;
      return c;
}
```

Figure 7–18. Short Temporaries

```
entry a1,32
add.n a2, a2, a3
slli  a2, a2, 16
srai a2, a2, 16
retw.n
```

Figure 7–19. Short Temporaries Assembly File

Much more efficient code would be generated if *c* was declared to be an *int*. In general, avoid using *short* and *char* for temporaries and loop variables. Try to limit their use to memory-resident data. An exception to this rule is multiplication. If you have 16-bit multipliers and no 32-bit multipliers, make sure that the multiplicands' data types are *short*.

7.7.4 Use of Globals Instead of Locals

Global variables carry their values throughout the life of a program. The compiler must assume that the value of a global might be used by calls or by pointer dereferences. Consider the example in Figure 7–20.

```
int g;
void foo()
{
      int i;
      for (i=0; i<100; i++){
           fred(i,g);
      }
}
```

Figure 7–20. Globals

Ideally, g would be loaded once outside of the loop, and its value would be passed in a register into the function `fred`. However, the compiler does not know that `fred` does not modify the value of g. If `fred` does not modify g, you should rewrite the code using a local variable as shown in Figure 7–21. Doing so saves a load of g into a register on every loop iteration.

```
int g;
void foo()
{
      int i, local_g=g;
      for (i=0; i<100; i++){
           fred(i,local_g);
      }
}
```

Figure 7–21. Replacing Globals with Locals

7.7.5 Use of Pointers Instead of Arrays

Consider a piece of code that accesses an array through a pointer such as in Figure 7–22.

```
for (i=0; i<100; i++)
              *p++ = ...
```

Figure 7–22. Pointers for Arrays

In every iteration of the loop, **p* is being assigned, but so is the pointer *p*. Depending on circumstances, the assignment to the pointer can hinder optimization. In some cases it is possible that the assignment to **p* changes the value of the pointer itself, forcing the compiler to generate code to reload and increment the pointer during each iteration. In other cases, the compiler cannot prove that the pointer is not used outside the loop, and the compiler must therefore generate code after the loop to update the pointer with its incremented value. To avoid these types of problems, it is better to use arrays rather than pointers as shown in Figure 7–23.

```
for (i=0; i<100; i++)
              p[i] = ...
```

Figure 7–23. Arrays Instead of Pointers

8. Hardware Overview and Block Diagram

Figure 8–24 shows a block diagram of the Xtensa LX7 microprocessor with its major blocks and its external connections. Most blocks have a number of configurable options allowing the Xtensa LX7 core to be tailored to a specific application. The type and size of local instruction and data memories, as well as the Xtensa Processor Interface (PIF) are also configurable. The designer can make application-specific extensions to the processors defining new register files and instructions using the TIE language. The following sections give details on the different configuration options, as well as details on the interfaces and protocols between the Xtensa core and the outside world.

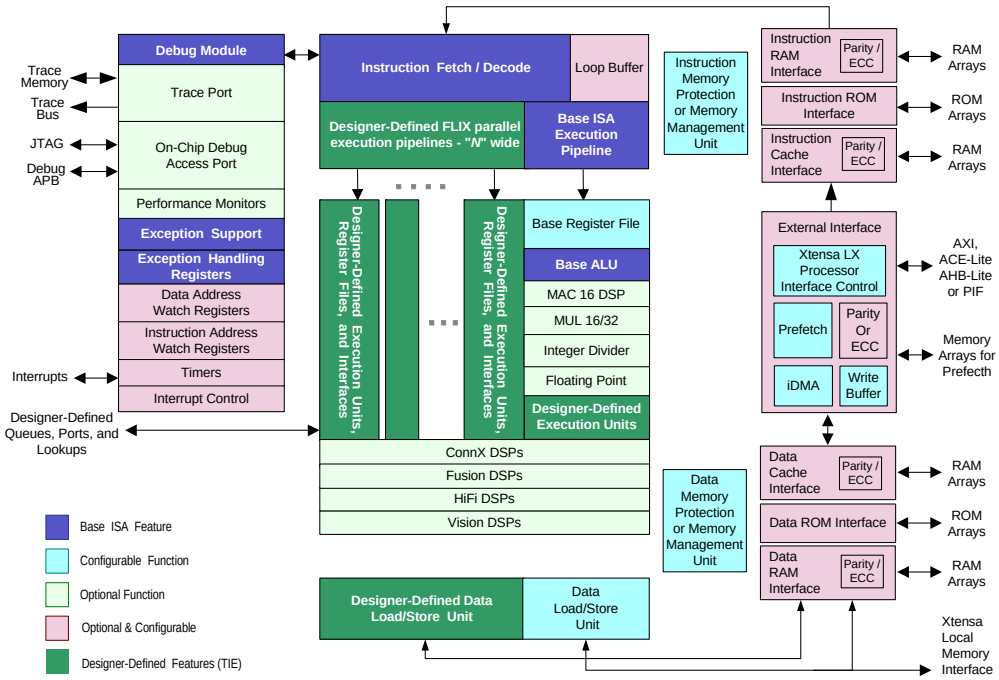


Figure 8–24. Configurable and Extensible Xtensa System

8.1 Design Hierarchy

The Xtensa LX7 processor's top-level module is named `Xttop`. The `Xttop` level of hierarchy contains the core of the processor along with the most common adjacent system-level components. `Xttop` includes the following components:

- Xtensa processor core

- Reset synchronizer
- Xtensa Debug Module
- AHB-Lite or AXI bridge
- PIF asynchronous FIFO to allow the AMBA bus and processor clocks to be asynchronous to each other. Requires use of the AHB-Lite or AXI bridge.

Figure 8–25 shows a high-level block diagram of Xttop and Xtmem. Note that the dotted lines indicate optional items.

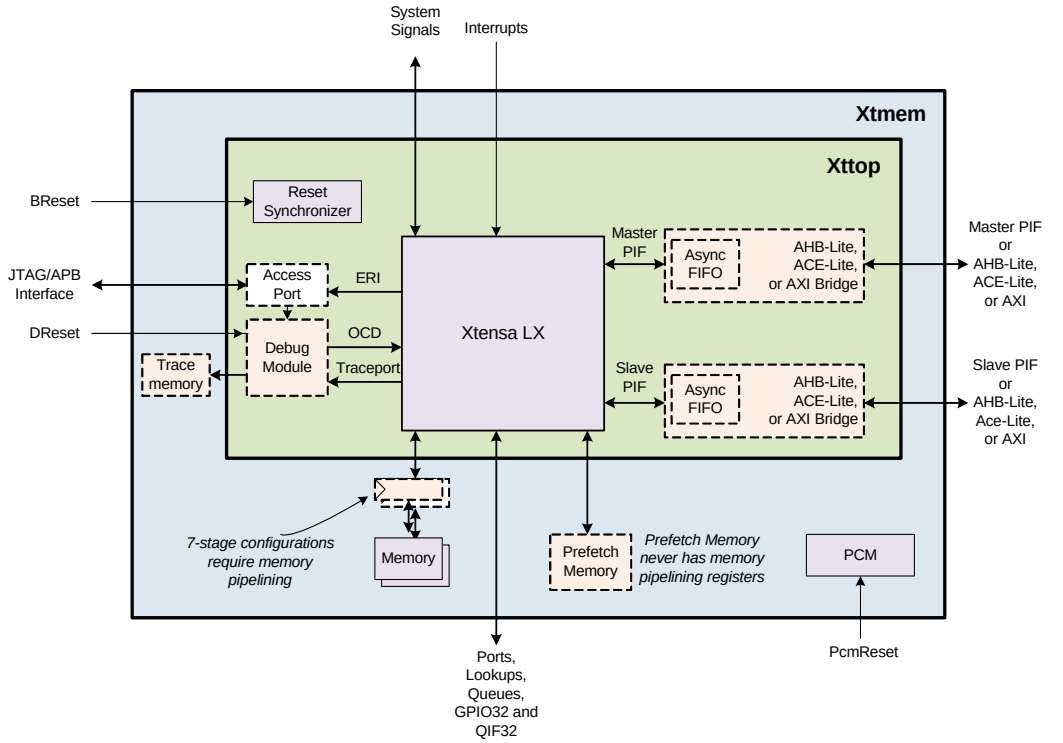


Figure 8–25. Xttop and Xtmem Block Diagram

The reference implementation flows and test bench use Xttop as the top level of hierarchy. Direct use of Xttop components, such as the Xtensa core, is discouraged and is not directly supported. RefTestbench always uses Xtmem as the top level of the hierarchy, but the local memory and cache instances are modeled with behavioral memory models until/unless you instantiate local memory cells and configure the test bench to use them.

8.2 The Xtensa LX7 Processor

The Xtensa LX7 processor core consists of a number of main blocks, including the core register files and execution units, the load/store units, the program-counter unit (PC unit) and instruction-fetch unit, the system-bus interface unit, trace and debug units, and a processor-control unit. Each unit offers a number of options that allow SOC designers to tailor them for a specific application and to meet area, power, and performance goals.

The register files and execution unit together implement the processor's general-purpose AR registers, special processor registers, and most of the instructions in the core instruction set architecture (ISA). General Xtensa instructions use four bits to identify the register for each source and destination operand, so these instructions access a 16-entry block of registers in the 32-bit AR register file. This 16-register block is either mapped directly to the 16 physical registers for processor configurations with 16-entry AR register files or mapped virtually through "register windowing" into AR register files with 32 or 64 entries. Register windowing can reduce the function-call overhead associated with saving and restoring registers.

The AR register file typically has two read ports and one write port, but additional read and write ports may be added if required for designer-defined TIE (Tensilica Instruction Extension language) extensions. The set of special registers present in a specific instance of the Xtensa LX7 processor core depends on the configuration options chosen.

Additional processor instructions can be added to the core architecture by including one of a number of predefined coprocessors and execution units (see Section 6.3), or designer-defined coprocessors and execution units using the TIE language. Optional and designer-defined coprocessors and execution units can use the existing general-purpose AR register file and can also define and use special, dedicated register files.

Load/store units implement the Xtensa LX7 load and store instructions. An Xtensa LX7 processor can have one or two load/store units. The processor's base ISA load and store instructions use the first load/store unit. The optional second load/store unit is used by TIE-based instruction extensions.

The load/store units connect to one of four types of local data memories. These local memories can include any of those listed in Table 8–12:

Table 8–12. Xtensa LX7 Processor Data-Memory Port Types and Quantities

Data-Memory Port Type	Minimum Number per Core	Maximum Number Per Core	Description
Data Cache Ways	0	4	Data caches may be direct-mapped or have 2, 3, or 4 ways. When multiple load/store units are present, the caches will be broken up into 2 or 4 banks.
Data RAM	0	2	Local data read/write memories, each data RAM can be configured as multi-banks (1, 2, 4)
Data ROM	0	1	Local data read-only memory, can be configured as multi-banks (1, 2, 4)
XLMI Port	0	1	High-speed local data/memory port
All	1	6	Total, all types of data memory ports

The Xtensa LX7 processor core can be configured with one or two load/store units, allowing as many as two load/store operations to occur in one instruction. The optional second load/store unit is available in conjunction with local-memory port options (data RAM and data ROM) as well as with data caches. However, certain configuration restrictions may apply. Each load/store unit has a separate interface port to each local memory and both ports must be connected to each configured local memory either through a memory-port arbiter or through the use of a true dual-ported memory design. In other words, none of the load/store unit ports can be left unconnected. Each local memory must be designed to interface to two processor ports for Xtensa LX7 configurations with two load/store units. The Xtensa LX7 Connection Box (C-Box) configuration option creates hardware to connect the two corresponding load/store-unit ports to each local memory, but not for the XLMI port.

The PC and instruction-fetch units control the fetching of instructions. The PC unit provides addresses to as many as four local instruction memories. The number and types of instruction memories are listed in Table 8–13:

Table 8–13. Xtensa Processor Instruction-Memory Port Types and Quantities

Instruction-Memory Port Type	Minimum Number per Core	Maximum Number Per Core	Description
Instruction Cache Way	0	4	Instruction caches may be direct-mapped or have 2, 3, or 4 ways
Instruction RAM	0	2	Local instruction read/write memories
Instruction ROM	0	1	Local instruction read-only memory
All	1	6	Total, all types of instruction-memory port

The instruction-fetch unit controls instruction-cache loading and feeds instructions to the execution unit. The basic Xtensa instruction size is 24-bits, with an optional fully compatible set of 16-bit instructions for improved code density. 16- and 24-bit instructions can be freely intermixed.

FLIX (Flexible-Length Instruction eXtensions) technology adds designer-defined 32- to 128-bit instructions in 8-bit increments to the Xtensa ISA. (An Xtensa LX7 configuration can have a choice of FLIX instructions sizes from 32 bits to 128 bits in 8-bit increments.) These wide-word options allow designers to create more complex, compound machine instructions to improve code and application performance. The wider instructions can be freely intermixed with the 24- and 16-bit Xtensa instructions.

The processor's external interface unit provides an interface to the processor interface bus (PIF). The PIF can be configured to different widths (32, 64, or 128 bits), and consists of separate, unidirectional input and output buses. The system-bus interface unit manages data transfers between the PIF and the processor's local instruction and data memory ports, as well as the XLMI port. In particular, this unit manages data and instruction cache-line requests and provides inbound-PIF (a PIF slave) capabilities to the processor's local instruction and data RAMs.

The control unit controls global operations such as reset, interrupts, and exception prioritization.

Finally, the Xtensa LX7 processor's trace and debug units, including the On-Chip Debug (OCD) module can be interfaced to a separate JTAG module (also provided by Cadence) for chip debug and test. The OCD module provides access to software-visible processor state. In addition, a separate trace port gives visibility into the Xtensa LX7 processor during real-time program execution.

8.3 Local Memories

The Xtensa LX7 processor has a number of local-memory interface options that allow fast access to instructions and data over a wide range of different applications. The four types of local-memory interfaces available are:

- Cache interface port (instruction and data)
- RAM interface port (instruction and data)
- ROM interface port (instruction and data)
- Xtensa Local Memory Interface (XLMI)

Figure 8–26 shows that local instruction memories can have a total of six arrays consisting of zero to four instruction-cache ways, zero to two instruction RAMs, and zero or one instruction ROM. Local data memories can have a total of six arrays consisting of zero to

four cache ways, zero to two data RAMs, zero or one data ROM, and zero or one XLMI interface. Each data RAM or ROM can be configured with multiple banks to improve load/store throughput. The number of banks can be 1, 2, or 4.

Memory error protection methods including parity and ECC are available on a local-memory-port by local-memory-port basis (excluding ROM ports and the XLMI port). Each local-memory port in a processor configuration can either be configured with the optional memory error protection or not. All data memories (including the data-cache ways) configured with memory error protection must have the same protection type (parity or ECC) and all instruction memories (including the instruction-cache ways) configured with memory error protection must have the same protection type.

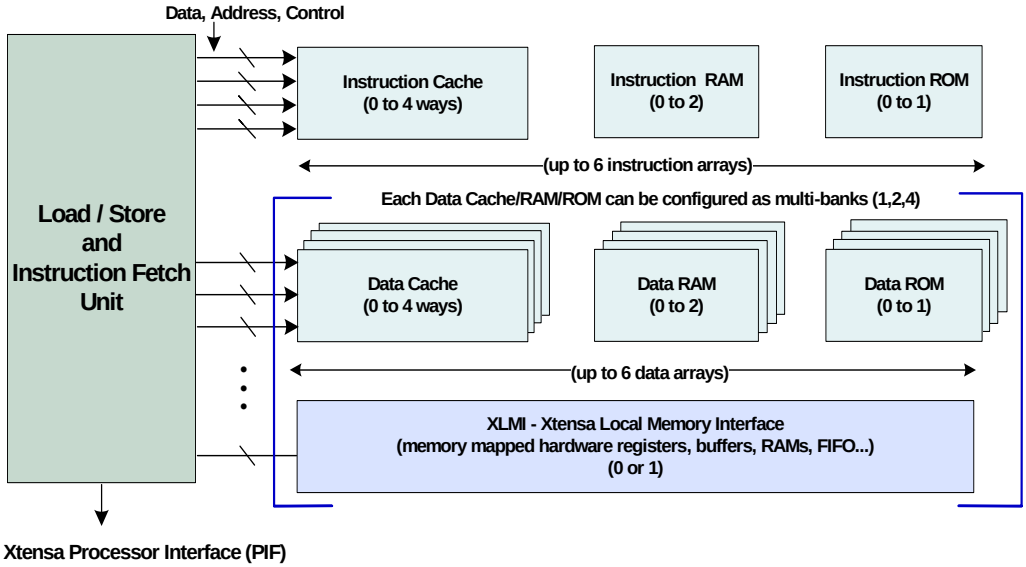


Figure 8–26. Local Memory Options

8.3.1 Caches

The Xtensa processor has separate data and instruction caches to significantly improve processor performance. Instruction and data caches can be configured to have different sizes (up to 128 Kbytes each), different line sizes (16, 32, 64, 128, or 256 bytes), and different associativities (1-4 ways) depending on the application requirements. The data cache can also be configured to be write-through or write-back. A cache-locking option allows cache lines to be fetched and locked down in the cache to avoid cache misses in critical code.

Note: The cache line-locking option requires more than one way of associativity depending on the configuration. The number of ways of cache must be greater than or equal to the number of load/store units + 1.

The Xtensa ISA provides instructions that allow the application code to manage and test the caches. These instructions allow the code to invalidate cache lines, directly read and write the cache arrays, and lock cache lines in the cache.

Each cache way actually consists of two memory arrays: one array for the data and one for the cache tag. Cache operation is described in detail in Chapter 15, "Xtensa Cache Interface Bus" on page 219.

8.4 ISA Implementation

This section defines the specific Xtensa LX7 implementation of some items that are undefined according to the Xtensa Instruction Set Architecture (ISA):

- The SIMCALL instruction is implemented as a NOP in hardware.

9. Configurable Hardware Features

The Xtensa processor core provides designers with a very wide range of configuration options and extensions. These configurable features allow SOC developers to mold the abilities and performance of the processor cores used on the SOC to precisely fit the requirements of the various application tasks performed by the SOC. The Xtensa LX7 processor's configurable options include:

9.1 Core Microarchitecture Options

This section lists the core microarchitecture options:

- 5- or 7-stage pipeline
  A 5-stage pipeline corresponds to a memory-access latency of one cycle. A 7-stage pipeline corresponds to a memory-access latency of two cycles. The default Xtensa LX7 pipeline has five stages: instruction fetch, register access, execute, data-memory access, and register writeback. This is a typical 5-stage RISC pipeline and it is sufficient for many uses on an SOC. However, as memories get larger (therefore slower) and logic gets faster, slow local memories have become a growing problem for SOC designs that use the most advanced IC processes to produce high processor clock rates.

  SOC designers using the Xtensa LX7 processor can compensate for mismatches between processor and memory speed by using this configuration option to extend the default 5-stage pipeline to seven stages, which adds an extra instruction-fetch stage and an extra memory-access stage. The extra stages can effectively double or triple the available memory-access time at high clock frequencies.
- Number of Load/Store Units
  1 or 2
- Instruction Fetch Width

  32-, 64-, or 128-bits. A wider instruction fetch allows the instruction fetch pipeline to fetch more instruction data per cycle. The Instruction Fetch Width is always bigger or equal to the maximum instruction size.
- Early Restart Option
  The pipeline can be configured for early restart on both instruction cache line misses and data cache line misses. For data, the instruction will wait in the M-stage of the pipeline and the data for the cache line miss data will be forwarded to it there. For instruction fetch, the fetch will stall in the I-stage and the cache line miss data will be forward to it there.Refill of the cache lines occurs in the background while the pipeline proceeds. Configurations that have two load/store units and data cache must choose Early Restart Option.

- Size of General-Purpose, 32-Bit Register File
  16, 32, or 64 entries
- Configurable Big- or Little-Endian Operation
- Unaligned Load/Store Action
  can perform in one of three different ways depending on the configuration option:
  - The align option forces an alignment by ignoring the lower bits of the address
  - The unaligned exception option causes an exception to be taken for unaligned load and store operations
  - The unaligned HandledByHardware option performs multiple memory accesses to sequential locations under the control of a hardware state machine in the processor. (Some complex and unusual cases of unaligned access will still result in taking an exception.)
- Synchronous or Asynchronous Reset
- Miscellaneous Special Register Option

  As many as four optional scratch registers can be configured. Software accesses these registers using the RSR, WSR, and XSR special-register instructions.
- Processor ID option

  Creates a processor ID (PRID) register that latches the state of the 16-bit PRID processor input bus during reset. This feature is useful for SoC designs with multiple copies of identical Xtensa processor configurations.
- Number of co-processors

  This is the highest number of co-processor number used in TIE source, then add one (highest number + 1).

9.2 Core Instruction Options

This section lists the core instruction options:

- Boolean

  This option makes a set of Boolean registers along with branches and other operations available. Coprocessors, such as the Cadence-designed floating-point coprocessor can use these Boolean architectural additions.
- Code Density
  The basic Xtensa instructions are encoded in either 16 or 24 bits for optimal code density. These 16- and 24-bit instructions can be freely intermixed in the instruction stream. The 16-bit instructions can be optionally deleted to save processor hardware at the expense of code density.

- Compound, Wide Instructions (FLIX)
  The designer can define compound FLIX instructions (Flexible Length Instruction Xtensions) in TIE and these instruction will be added to the processor by the Xtensa Processor Generator. Any number of instruction sizes can be defined, ranging from 32- to 128-bit instructions, in 8-bit (1 byte) increments. The additional widths can be freely intermixed with 24- and 16-bit instructions.
- Zero-Overhead Loop (ZOL)
  The ZOL option adds special loop instructions and a loop-iteration counter, which is automatically decremented by the instruction-fetch unit, to the Xtensa ISA. The ZOL option speeds loop execution and reduces code space with respect to conventional software loops, which employ explicit instructions within the loop to decrement a loop counter and an explicit branch instruction to initiate another loop iteration. The Xtensa processor's zero-overhead-loop instructions and logic can be removed to save area when the extra performance provided by the ZOL option is not needed for a target application.

  Note: Removing the zero-overhead-loop instructions from the Xtensa ISA can cause binary code incompatibility with certain commercial 3rd-party software such as real-time operating systems.
- MAC16
  This option adds a 16x16-bit multiplier and a 40-bit accumulator, eight 16-bit (or four 32-bit) operand registers, load instructions for the additional operand registers, and a set of compound instructions that combine multiply, accumulate, load, and automatically update operand addresses.
- MUL16
  This option adds a 16x16-bit multiplier that produces a 32-bit result.
- MUL32
  This option adds a 32x32-bit multiplier that produces a 64-bit result. The Mul32High option controls whether the upper 32-bits of the 64-bit result are made available to the software. If only integer multiplication is required, the MUL32 option can be implemented with reduced area by selecting the iterative implementation of the MUL32 option.
- DIV32
  This option adds a 32-bit integer divider that supports signed and unsigned division and remainder operations. The quotient and remainder divide instructions are implemented iteratively and the divide operation latencies vary depending on the size of the quotient.
- Synchronization
  This option implements load-acquire and store-release instructions that permit the management of load/store operation ordering.

- Conditional Store Synchronization Instruction
  This option implements the store conditional instruction, S32C1I, which is used for updating synchronization variables in memory shared by multiple interrupt handlers, multiple processes, or multiple processors.
- Exclusive Store Instructions
  Exclusive access instructions can create semaphore and other exclusion operations. Unlike regular stores, the exclusive store (S32EX) instruction is conditional and updates memory only if no master has performed a store to the same location following an exclusive load (L32EX). The GETEX instruction allows the processor to read the EXCLRES (exclusive result) register and CLREX clears the EXCLMON (exclusive access monitoring operation) register set by a previous L32EX instruction. See Section 4.7” for details.
- Miscellaneous Instructions
  These options implement miscellaneous instructions including Normalization Shift Amount instructions (NSAU, NSA), Minimum and Maximum Value instructions (MIN, MAX, MINU, MAXU), the Clamp instruction (CLAMP), the Sign Extend instruction (SEXT), and the Deposit Bits (DEPBITS) instruction.
- Thread Pointer
  The Thread Pointer Option provides an additional register to facilitate implementation of Thread Local Storage for use by operating systems and software-development tools. See the Thread Pointer Option section in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for details.
- SuperGather™ option
  SuperGather operations improve the efficiency of SIMD operations using gather/scatter operations. Gather operations load data from arbitrary addresses in local memory into subsequent lanes of a SIMD register. Scatter operations take data from consecutive lanes of a SIMD register and store it to non-contiguous addresses in local memory.

  The SuperGather option must be used in conjunction with Vision P5 or Vision P6. The core option adds registers to control and report errors associated with SuperGather operations. Consult the *Vision P5 User’s Guide* and *Vision P6 User’s Guide* for details.

9.3 Optional Function Units

This section lists the Xtensa processor optional function units:

- IEEE 754 Floating-point Unit (FPU)
  This 32-bit single precision floating point coprocessor option is optimized for printing, graphics, and audio applications. An IEEE 754 compatible FPU, this co-processor provides the programming ease of floating-point arithmetic with the execution speed of fixed-point processing.
- IEEE 754 Double Precision Floating-point Extension (DFPU)
  This extends the IEEE 754 Floating Point Unit (FPU) to handle 64-bit double precision operations as well as 32-bit single precision floating point operations. Double precision operations have greater precision and greater range than single precision and operate with about the same performance.
- Double-Precision Floating-Point Accelerator
  The Xtensa double-precision floating-point accelerator adds a small set of TIE instructions and states to speed existing double-precision software execution of IEEE 754 double-precision floating-point calculations. This option implements IEEE 754 compliant 64-bit double-precision add, subtract, multiply, square root, and divide operations with the round-to-nearest rounding mode. It also implements a complete set of comparison operations that allow for IEEE-compliant comparisons.
- GPIO32
  The GPIO32 option provides a preconfigured output and input ports that allow SOC designers to integrate the processor core more tightly with peripherals.
- QIF32
  The QIF32 option provides preconfigured input/output queue interfaces. These queue interfaces, which can be used to connect to standard FIFOs, provide a useful mechanism for the processor to communicate with other RTL blocks, devices, and even other processors, without ever using the system bus.
- FLIX3
  The FLIX3 option selects a pre-configured 3-way FLIX Xtensa CPU core, along with Xtensa instructions allocation per FLIX slot. With this option, three Xtensa base ISA instructions, each operating on separate 32-bit data, can be issued simultaneously in 64-bit FLIX instruction formats.The FLIX instructions allow multiple core instructions to be executed at the same time.

9.4 DSP Options

This section lists the Xtensa processor DSP options:

- ConnX BBE16EP DSP
  The ConnX BBE16EP (16-MAC Baseband Engine) is based on a ultra-high performance DSP architecture designed for use in next-generation baseband processors for LTE Advanced, other 4G cellular radios and multi-standard broadcast receivers. The high computational requirements of such applications require new and innovative architectures with a high degree of parallelism and efficient I/Os. The ConnX BBE16EP meets these needs by combining a 8-way Single Instruction, Multiple Data (SIMD), 16 multiplier accumulators (MAC) and up to 5-issue Very Long Instruction Word (VLIW) processing pipeline with a rich and extensible set of interfaces.
- ConnX BBE32EP DSP
  The ConnX BBE32EP (32-MAC Baseband Engine) is based on a ultra-high performance DSP architecture designed for use in next-generation baseband processors for LTE Advanced, other 4G cellular radios and multi-standard broadcast receivers. The high computational requirements of such applications require new and innovative architectures with a high degree of parallelism and efficient I/Os. The ConnX BBE32EP meets these needs by combining a 16-way Single Instruction, Multiple Data (SIMD), 32 multiplier accumulators (MAC) and up to 5-issue Very Long Instruction Word (VLIW) processing pipeline with a rich and extensible set of interfaces.
- ConnX BBE64EP DSP
  The ConnX BBE64EP (64-MAC Baseband Engine) is based on a ultra-high performance DSP architecture designed for use in next-generation baseband processors for LTE Advanced, other 4G cellular radios and multi-standard broadcast receivers. The high computational requirements of such applications require new and innovative architectures with a high degree of parallelism and efficient I/Os. The ConnX BBE64EP meets these needs by combining a 32-way Single Instruction, Multiple Data (SIMD), 64 multiplier accumulators (MAC) and up to 5-issue Very Long Instruction Word (VLIW) processing pipeline with a rich and extensible set of interfaces.
- HiFi 2 Audio Engine
  The Xtensa HiFi 2 Audio Engine provides a fully optimized approach to embedding multiple audio standards into SOC designs. It is optimized to run all of the available audio codecs in a pre-packaged, drop-in format with all software components ported. The HiFi 2 Audio Engine's performance can approach that of hardware implementations, while providing better flexibility, risk reduction, time to market. It reduces silicon area and power consumption by running all codecs on one hardware block.
- HiFi EP Audio Engine (requires the HiFi 2 Audio engine)
  The HiFi EP configuration option extends the HiFi 2 Audio Engine ISA with support for predictive hardware prefetching, 32x24-bit multiply/accumulate operations, circular buffer loads and stores and bidirectional shifts.

- HiFi Mini
  The HiFi Mini Audio Engine is a highly optimized audio processor geared for efficient execution of audio and voice codecs, pre- and post-processing modules, and low-power applications such as voice trigger. The HiFi Mini Audio Engine is a configuration option that can be included with the Xtensa LX processor. All HiFi Mini Audio En-gine operations can be used as intrinsics in standard C/C++ applications. Our software development tool chain is fully aware of all HiFi Mini operations included in the source code, and they are dispatched to the corresponding hardware functional units.

- HiFi 3 DSP
  The HiFi 3 DSP is a highly optimized audio processor geared for efficient execution of audio and voice codecs and pre- and post-processing modules. It goes beyond the two MAC, two issue, HiFi 2/EP architecture with four multipliers, three VLIW slots, good support for 32x16-bit and 32x32-bit multiplication, a true 64-bit data path and native support for ITU-T/ETSI intrinsics.

- HiFi 4 DSP
  The HiFi 4 DSP is a highly optimized audio processor geared for efficient execution of audio and voice codecs and pre- and post-processing modules. It goes beyond HiFi 3 with support for four 32x32-bit MACs, some support for 72-bit accumulators, limited ability to support eight 32x16-bit MACs, a fourth VLIW slot and the ability to issue two 64-bit loads per cycle. There is an optional floating point unit available, providing up to four single-precision IEEE floating point MACs per cycle. The extra resources provide for significant performance improvements compared to HiFi 3, particularly on pre/post-processing algorithms.

- HiFi 3z DSP
  The HiFi 3z DSP, is a highly optimized audio processor geared for efficient execution of audio, and voice codecs, and pre-, and post- processing modules. It goes beyond HiFi 3, with the ability to issue two 64-bit loads per cycle, two Xtensa core operations per cycle, and limited ability to support eight, 16x16-bit MACs per cycle (including specialized instructions to accelerate 16-bit FFT performance). The HiFi 3z DSP supports 40-bit/48-bit instruction formats for potential code size reduction, and instructions to accelerate arithmetic decoding. An optional floating point unit available, supporting a 2-way SIMD, single-precision IEEE floating point MAC or ALU operation, every cycle. The extra resources help achieve significant performance improvement compared to HiFi 3, particularly on pre/post processing algorithms and voice, and audio codecs.

- Vision P5 DSP
  The Vision P5 DSP is a high-performance, low-energy engine designed for image and computer vision processing. It implements 64-way, 8-bit SIMD with five VLIW slots. This DSP also offers a 1024-bit memory bus and a comple subsystem using the integrated DMA (iDMA) option for transferring high-resolution data. Vision P5 also supports a16-way, single precision (32 bit) vector floating-point unit.

- Vision P6 DSP
  The Vision P6 vision and imaging DSP is a high performance embedded digital signal processor (DSP) optimized for vision, image, and video processing. The instruction set architecture and memory subsystem provide easy programmability in C/C++ and deliver the high sustained pixel processing performance required for advanced vision, image, and video processing and analysis applications.

  The Vision P6 is a vector (SIMD) DSP supporting 64-way 8-bit, 32-way 16-bit, and 16-way 32-bit signed and unsigned integer and fixed-point types and operations. The Vision P6 optionally supports 16-way single-precision and 32-way half-precision floating-point types and operations.
- Fusion F1 DSP
  The Fusion F1 DSP is a highly optimized, highly configurable, processor geared for efficient execution of dataplane algorithms needed for the Internet of Things (IoT), and other applications, such as codec chips, sensor hubs, and narrowband wireless communications.

  The Fusion F1 DSP is derived from a smaller version of the HiFi 3 DSP. It supports dual issuing a single load or a store together with two way SIMD ALU or MAC operations, supporting dual 16x16, 32x16, 24x24-bit MACs and single 32-bit MACs. The base configuration is source code software compatible with the HiFi 2, HiFi Mini and HiFi 2 EP Audio Engines except for bit-stream and variable-length decode and encode. It is also compatible with the HiFi 3 DSP except for bit-stream, variable-length decode and encode and HiFi 3 quad MAC instructions.
- Fusion G3 DSP
  The Fusion G3 (Vector DSP) is based on an ultra-high performance DSP architecture designed for use in high-performance general-purpose DSP applications. The Fusion G3 supports vectorization for all standard C datatypes. This includes 8-bit, 16-bit, 32-bit, 64-bit signed and unsigned integer types. It also includes single-precision and double-precision floating point types. In addition, vectorization of C99 complex floating point types is supported.

  The Fusion G3 DSP combines a multiple-type Single Instruction, Multiple Data (SIMD) datapath with integral and floating point multiply-accumulate (MAC) units and Very Long Instruction Word (VLIW) processing pipeline with a rich and extensible set of interfaces.

  Fusion G6 DSP
  The Fusion G6 (Vector DSP), is based on ultra-high performance DSP architecture, designed for use in high-performance, general-purpose DSP applications. The Cadence Fusion G6 supports vectorization for all standard C datatypes. This includes 8-bit, 16-bit, 32-bit, 64-bit signed, and unsigned integer types. It also includes single precision and double-precision floating point types.

The Fusion G6 DSP additionally supports, vectorization of C99 complex floating point types. It combines, a multiple-type Single Instruction, Multiple Data (SIMD) datapath, with integral and floating point multiply-accumulate (MAC) units, and a Very Long Instruction Word (VLIW) processing pipeline, with a rich and extensible set of interfaces.

9.5 Exception and Interrupt Options

The Xtensa LX7 processor supports the Xtensa Exception Architecture 2 (XEA2), which supports ring protection (in processor configurations that include the MMU option) and simplifies the creation of exception handlers. The following exception options are available:

- Relocatable Vectors Option
  The Relocatable Vectors Option allows vectors to be moved dynamically, as a group. Vectors are divided into two groups based on the relocation mode.

  The first vector group, called the Static Vectors group, includes the reset and memory-error-handler vectors. These vectors reside, by default, in one of two statically determined locations based on the state of the `StatVectorSel` input pin just before the processor exits the reset state. There is also a sub-option under this Relocatable Vectors option, called the External Reset Vector option. This sub-option adds an input bus to the processor interface pins, which allows the user to provide an alternate reset vector. For more information, see Section 22.4 “Static Vector Select” on page 416.

  The second vector group, called the Relocatable Group, gives each vector a fixed offset from a base address. The base address is held in a special register, `VECBASE`, which is initialized to a preconfigured value during reset. However, software can change the value of the vector base register, thereby moving the Relocatable Group vectors within the processor’s address space.
- Interrupt Option
  This option adds the ability to handle as many as 32 asynchronous exceptions generated from external and certain internal sources. The interrupt option on its own implements Level-1 interrupts, which are the lowest priority interrupts. All level-1 interrupts share one user- and one supervisor-level interrupt vector.
- Number of Interrupts
  Configurable, 1 to 32. The total number interrupts is equal to the number of Level-1 interrupts plus the number of high-priority, non-maskable, and timer interrupts. Note that instruction exceptions are decoupled from the option that sets the number of available interrupts and are not a factor in the 32-interrupt limit.

- Interrupt Types
  - *External Level*—Level-sensitive input interrupt signal to the processor.
  - *External Edge*—Edge-triggered input interrupt signal to the processor.
  - *Internal*—An interrupt generated by internal processor logic (for example: timer and debug interrupts).
  - *Software*—An interrupt generated by software manipulation of read/write bits in the processor's `INTERRUPT` register.
  - *Non-maskable (NMI)*—External, edge-triggered interrupt input signal to the processor with an implicitly infinite priority level.
  - Write-Error Interrupt - The PIF Write Response Option creates an interrupt source that generates an interrupt for a write-response error.
  - Profiling—The Performance Monitor Option creates a level-sensitive input to the processor (from the Debug Module), when a performance counter overflows.
- iDMA Interrupts

  The Integrated DMA engine can be configured with interrupts that notify the Xtensa processor if a particular descriptor finishes or when an error occurs.
- Gather/Scatter Interrupts
  The SuperGather engine can be configured with interrupts that notify the Xtensa processor if a particular non-precise error occurs. See the *Vision P5 User's Guide* and Vision P6 User's Guide for detailed information.
- High-Priority Interrupts and Non-Maskable Interrupt
  These options implement a configurable number of interrupt levels (with a maximum of six) and an optional non-maskable interrupt (NMI). Each of the optional high-priority interrupts and the one optional non-maskable interrupt have individual interrupt vectors.
- C-callable Interrupts
  The `EXCMLEVEL` parameter, part of the high-priority interrupt option, allows the designer to designate the number of high-priority interrupts that are C-callable.

  Note: Level-1 interrupt handlers may always be written in C.
- Timer Interrupts
  This option implements from one to three internally-generated timer interrupts based on special registers that contain interrupt-trigger values. The processor compares the values in these registers with a 32-bit counter that is incremented each processor clock cycle. Interrupts occur when the values in the comparison and counter registers match and the timer interrupt is not masked.
- Interrupt Vectors
  High priority and non-maskable interrupts have individual interrupt vectors per level.

9.6 Memory Management Options

The Xtensa processor has the following memory management options:

- Region Protection
  This configuration option has an instruction Translation Lookaside Buffer (ITLB) and a data Translation Lookaside Buffer (DTLB) that divide the processor's address space into eight memory regions. These Translation Lookaside Buffers (TLBs) provide region protection for memory. The access modes of each memory region can be changed by writing to the appropriate TLB entries.

  Note: With region-protection alone, the Xtensa TLBs do not translate addresses (memory addresses are always identity mapped through the TLBs).
- Region Protection with Translation
  This configuration is similar to the Region Protection option but it adds virtual-to-physical address translation for the eight memory regions.
- MMU with TLB and Autorefill
  The MMU with TLB configuration option is a general-purpose MMU that supports the memory-management needs of operating systems such as Linux and Windows CE. It provides facilities for demand paging and memory protection.
- MPU

  The Memory Protection Unit (MPU) provides memory protection with a smaller footprint than the Linux-compatible MMU, but with much more flexibility and features than the Region Protection Unit.

9.7 Cache and Cache Port Options

The Xtensa processor core offers the following instruction- and data-cache options:

- Data Cache
  0, 1, 2, 4, 8, 16, 32, 64, and 128 Kbytes (for 1 or 2-way set associative)
  0, 1.5, 3, 6, 12, 24, 48, and 96 Kbytes (3-way set associative)
  0, 2, 4, 8, 16, 32, 64, and 128 Kbytes (for 4-way set associative)
- Instruction Cache
  0, 1, 2, 4, 8, 16, 32, 64, and 128 Kbytes (for 1 or 2-way set associative)
  0, 1.5, 3, 6, 12, 24, 48, and 96 Kbytes (3-way set associative)
  0, 2, 4, 8, 16, 32, 64, and 128 Kbytes (for 4-way set associative)
- Cache Line Size
  16, 32, 64, 128, or 256 bytes
  When data cache banking is configured (banking only available on the Data Cache), the cache line size must be at least the width of all the banks

- Cache Port Width
  The Xtensa processor's data cache buses can be configured to be 32-, 64-, 128-, 256-, or 512-bits wide and the instruction cache buses can be configured to be 32-, 64-, or 128-bits wide
  - The Xtensa processor's instruction- and data-cache ports can be configured to have different widths.
  - The processor's cache-memory ports and PIF can be configured to have different widths.

    (**Note:** Xtensa processors configured with caches must also be configured with a PIF, which permits main-memory access to service cache refill and castout operations.)
  - The processor's instruction-cache port width should be set equal to the instruction-fetch width to minimize the power dissipation associated with instruction-fetch operations (see Section 9.19).
- Cache-Memory Access Latency
  1- or 2-cycle access latency for 5-stage and 7-stage pipelines respectively
- Associativity and Cache-Line Refill Method
  Direct-mapped or 2, 3, or 4-way set-associative cache
  Least-recently-filled (LRF) per cache-line index refill algorithm
  At least 2 ways of cache must be configured for two load/store units
  At least 3 ways of cache must be configured for two load/store units and line locking
- Data-Cache Write Policies
  Write-Through
  Write-Back
- Cache-Line Locking (for 2, 3, or 4-way set-associative caches)
- Data Cache Banking (1, 2, or 4 banks)
  More than one bank is only supported with two load/store units. Each bank of memory must be one load/store unit width wide. Two load/store unit configurations require the early restart option and at least 2-way set associative caches.
- Optional Cache Memory ECC/Parity Support
  Allows the caches to be protected against errors using either parity or error-correcting code (ECC). The Xtensa processor uses a SECDED (single-error correction, double-error detection) ECC.
- Optional Prefetch
  A prefetch block allows cache lines to be prefetched into the processor whenever sequential accessing of data or instruction cache lines is detected. This option requires a data cache. While prefetch always applies to the data cache, it can also apply to the instruction cache in many cases. The details for when Prefetch applies to the instruction cache are listed in Section 17.2.2.

- Number of Cache Prefetch Entries
  The number of prefetch entries can be configured to be 8 or 16 entries. This determines how many cache lines can be being prefetched at one time.
- Prefetch to L1 Data Cache Option
  The prefetch logic can be configured so that it can optionally put data cache prefetch responses directly into the L1 Data Cache, rather than only bringing it into the prefetch buffer RAM. This can completely eliminate certain data cache misses if data is prefetched soon enough, but requires substantially more logic. Prefetch to L1 Data Cache Option requires an early restart pipeline (option).The Prefetch to L1 Data Cache Option requires an early restart pipeline, as well as an extra auxiliary tag array so that prefetch lookups can occur in parallel with pipeline cache accesses.
- Prefetch Castout Cache Line Buffers
  If writeback caches are configured and prefetch directly to L1 data cache is configured, then the prefetch logic may need to cast out cache lines, in which case it will need buffering to store these castouts. This buffering can be one or two cache lines large.
- Optional Block Prefetch
  Configurations with Prefetch to L1 Data Cache can optionally configure Block Prefetch. This allows Block Prefetch instructions to specify ranges of addresses that are to be prefetched (upgraded) or castout/invalidated (downgraded) in the L1 data cache as a background operation. There are eight Block Prefetch Entries, allowing up to eight Block Prefetch operations to be active at once.
- Dynamic Cache Way Usage Control
  The number of cache ways in use can be changed dynamically with this feature. It gives the ability to disable and re-enable the use of cache ways in both Instruction-Cache and Data-Cache independently to facilitate power savings. New and modified instructions enable the user to clean cache ways before disabling them and to initialize cache ways while enabling them. When a Cache Way is disabled, it removes that cache memory block from service. Therefore it reduces total cache capacity by 1/(number of ways in service).

9.8 RAM Port Options

The Xtensa processor core offers the following RAM port options:

- 0, 1, or 2 data RAMs
  sizes of 0.5, 1, 2, 4, 8, 16, 32, 64, 128, and 256 Kbytes
  (By request, the maximum configurable size of the local data memories can be increased to 512 kbytes or to 1, 2, or 4 Mbytes.)
- 0, 1, or 2 Instruction RAMs
  sizes of 0.5, 1, 2, 4, 8, 16, 32, 64, 128, and 256 Kbytes
  (By request, the maximum configurable size of the local instruction memories can be increased to 512 kbytes or to 1, 2, or 4 Mbytes)

- Data RAM Access Width
  The data RAM ports can be configured to be 32-, 64-, 128-, 256-, or 512-bits wide. All data RAM ports have the same width in a configuration.
- Instruction RAM Access Width
  The instruction-RAM port width can be configured to be 32, 64, or 128 bits. All instruction-RAM ports have the same width in a configuration. The port width must match the configured instruction-fetch width (see Section 9.19).
- Data RAM Banks
  The Xtensa processor core can directly connect to single-ported data RAMs. The data RAM can be configured as 1,2, or 4 banks of single-ported RAMs. Data RAMs must have the same number of banks. The Xtensa processor core cannot directly connect to dual-ported data RAMs. To use dual-ported data RAMs, an external customer-designed Connection Box (C-Box) can be designed to bridge the core to the RAMs.
- RAM Port Latency
  1- or 2-cycle access latency for 5-stage and 7-stage pipelines respectively.
- RAM Busy
  Data- and instruction-RAM ports have optional busy inputs that allow these local memories to be shared among multiple processors or initialized externally.
- Inbound-PIF Operations to RAM
  Both data- and instruction-RAM ports optionally permit inbound-PIF read and write operations to the attached local RAMs.
- Base Physical Address
  The starting physical address for each RAM is configurable. The starting address must be aligned to the RAM size.

9.9 ROM Port Options

The Xtensa processor core offers the following ROM port options:

- 0 or 1 Data ROM
  0.5, 1, 2, 4, 8, 16, 32, 64, 128, and 256 Kbytes
  (By request, the maximum configurable size of the local data memories can be increased to 512 kbytes or to 1, 2, or 4 Mbytes.)
- 0 or 1 Instruction ROM
  0.5, 1, 2, 4, 8, 16, 32, 64, 128, and 256 Kbytes
  (By request, the maximum configurable size of the local instruction memories can be increased to 512 kbytes or to 1, 2, or 4 Mbytes.)

- Data ROM Port Width
  The data-ROM port can be configured to be 32-, 64-, 128-, 256-, or 512-bits wide.
- Data ROM Banks
  The data ROM can be configured for 1, 2, or 4 banks. For configurations with two load/store units, multiple banks must be used together with C-Box.
- Instruction ROM Port Width
  The instruction ROM port width can be configured to be 32, 64, or 128 bits. The port width must match the configured instruction-fetch width (see Section 9.19).
- ROM Latency
  1- or 2-cycle access latency for 5-stage and 7-stage pipelines respectively.
- ROM Busy
  Both data- and instruction-ROM ports have optional busy inputs that allow these local memories to be shared among multiple processors or initialized externally.
- Base Physical Address
  The starting physical address for each ROM is configurable. The starting address must be aligned to the ROM size.

9.10 Integrated DMA

The Xtensa processor can be configured with an intergrated DMA engine (iDMA). iDMA allows you to configure the following parameters:

- Configurable bus buffer depth (1/2/4/8/16): Adding more bus buffers provides better capability to off-load the bus traffic; this reduces the chance of iDMA running out of buffers, which can cause a temporary bus deny. This option allows trade off between area and performance.
- Configurable number of outstanding 2D rows (2/4/8/16): Allowing more outstanding 2D row requests improves the iDMA bus performance at the cost of area. This option allows trade off between area and performance.

9.11 Xtensa Local Memory Interface (XLMI) Port Options

The Xtensa processor core offers the following XLMI port options:

- One optional XLMI port, mapped to an address space of size:
  0.5, 1, 2, 4, 8, 16, 32, 64, 128, and 256 kbytes.
  (By request, the maximum configurable size of the XLMI port's address space can be increased to 512 kbytes or to 1, 2, or 4 Mbytes.)

- XLMI Port Width
  The XLMI port can be 32-, 64-, or 128-bits wide, as set by the local-data-memory port width configuration setting.
- XLMI Port Latency
  1- or 2-cycle access latency for 5-stage and 7-stage pipelines respectively.
- XLMI Busy
  The XLMI port has an optional busy input that allows local memory or other devices attached to this port to be shared among multiple processors or initialized externally.
- Base Physical Address
  The starting physical address for the XLMI port is configurable. The starting address must be aligned to the size of the address space allotted to the XLMI ports.
- Handling speculative loads
  Unlike the processor's other local-memory ports, the XLMI port is a high-speed port designed to connect to blocks and devices other than memory. As with the processor's local-memory ports, speculative operations will appear on the XLMI port. While RAM and ROM blocks do not suffer from speculative-read effects, other blocks such as FIFO memories and most I/O devices do. Consequently, the XLMI port has signals to indicate when a load has been retired to help ensure that speculative-read effects do not cause improper operation of devices attached to the XLMI port.

Note: XLMI can not be used in configurations that have two load/stores.

Note: Inbound PIF does not apply to the XLMI port.

9.12 C-Box (Connection Box) Option

There is a Connection Box (C-Box) option for two-load/store configurations; either true or false can be selected. This option gives customers a choice if they want to design their own local memory arbiter. Note that this choice is only for two-load/store configurations. It is not available for single load/store configurations. Single-load/store configurations always have internal arbiter.

If C-Box option is *true* for two-load/store configurations, it offers connectivity between multiple load/store units and multiple single-ported local memories (the data RAMs and the data ROM) by providing all the logic necessary to funnel the processor's multiple

ports for each local memory into one interface per memory. To improve data RAM/ROM access throughput, the C-Box not only provides load/store sharing but also enables the Xtensa core to be connected to multiple-bank (up to four) data RAM or data ROM.

If the C-Box option is *false* for two-load/store configurations, the data RAM/data ROM interface for each load/store unit will be exported to be used by a customer-designed external arbiter. If the inbound PIF option is configured for that memory, an additional set of read/write ports for DMA will be generated.

Note: Setting the C-Box option to *false* for two load/store configurations exports dedicated data RAM read and write interfaces called split read/write ports. It no longer supports direct connection to dual-ported memories (see Section 18.10 “Exposed Processor Interface for a Customer-Designed C-Box” for details).

The C-Box is especially useful for DSP applications that require an XY memory configuration. The Xtensa processor’s local data RAM 0 would be considered the X memory, which is accessed using FLIX slot-0 load and store operations. Local data RAM 1 would be considered the Y memory, which is accessed from the alternate FLIX slot’s load and store operations. Thus local data RAM 0 is considered the primary RAM array for the processor’s first load/store unit and local data RAM 1 is the primary RAM array for the second load/store unit. The C-Box provides a single access port for each local memory that allows the configuration to operate in a "safe" mode.

An "unsafe" method of attaching X and Y memories to the processor would be to connect only the load/store ports of interest to the relevant memory arrays while leaving the other load/store ports unconnected. Such an "unsafe" configuration might work for some applications but there is no mechanism—an exception, for example—to protect the processor from errant memory references to the "wrong" RAM from the "wrong" FLIX operation slot. The C-Box makes such a mechanism unnecessary because it safely handles all memory accesses. (Note that accesses to a non-primary RAM array through the C-Box require additional time due to contention for that RAM array.)

Restrictions on C-Box are:

- C-Box is only applicable in two-load/store configurations. All data RAM/Data ROM must have the same C-Box choice.
- In two-load/store configurations, when C-Box is false, banks must 1 and busy should be configured for that memory. This allow customers to design their own external arbiter.
- C-Box must be true for multi-banked (2,4) data RAMs/Data ROMs in two-load/store configurations.

9.13 System Memory Options

The processor has the following system memory options:

- System ROM Option
  One System ROM can be optionally configured. A size and base physical address must be specified.
- System RAM Option
  One System RAM can be optionally configured. A size and base physical address must be specified.

The Xtensa Processor Generator uses the System ROM and RAM options to set up a default memory map, which affects the software-linker and hardware-diagnostic scripts automatically created by the generator. Defining the size and base addresses for the system memories also creates a default memory map that can run out-of-the-box C applications.

9.14 Memory Parity and ECC Options

The parity error-checking configuration option allows an Xtensa processor to detect memory errors. The ECC (error-correcting code) configuration option allows an Xtensa processor to detect and correct memory errors. Both options can cause program exceptions when an error is detected. The ECC option corrects single-bit data-access errors on the fly without causing an exception and detects double-bit data-access errors.

Two major types of memory-data errors can occur:

- Hard errors—These errors are permanent and usually result from broken wiring, shorts or open circuits.
- Soft errors—These errors are transient and can be caused by alpha particles, noise, and/or power surges.

Hard memory errors can only be corrected by physically repairing the damaged circuit. Soft memory errors can be corrected by storing the correct data back into memory.

The Xtensa processor core supports two types of memory-error protection:

- Parity—single-bit error detection. The parity option allows the processor to detect single-bit errors. It cannot detect the position of the bit in error.
- ECC—single-bit error detection and correction, double-bit error detection (SECDED). The Xtensa processor's optional ECC scheme can correct single-bit errors and can detect double-bit errors in local instruction and data RAMs and the instruction and data caches' tag and data arrays.

Notes:

- The error-checking options only apply to the Xtensa processor's local RAMs and to the data and tag arrays in the processor's instruction and data caches.

- One configuration option selects the memory-protection type (parity or ECC) for all data RAMs and the data and tag arrays in the data cache. Another configuration option selects the memory protection type for all instruction RAMs and the data and tag arrays in the instruction cache. In addition, the selected memory-protection option can be individually configured on or off for each local RAM type and cache memory type. Finally, a separate option selects protection on the prefetch RAM.
- The memory-protection options are not supported on the Xtensa processor's local ROM ports, on the XLMI port, or on the processor interface (PIF) bus.

9.14.1 Parity Error Checking

When the parity configuration option is selected, the Xtensa processor appends one parity bit to the end of a data item. For data memories such as data RAM or a data cache's data array, the parity option can be configured to either add one parity bit per byte of data or one parity bit per 32-bit word of data.

The Xtensa memory-parity option uses even parity.

9.14.2 SECDED ECC

The Xtensa ECC implementation uses minimum odd-weight-column SECDED codes that detect and optionally correct single-bit errors and detect double-bit errors. For data memories, there are 2 available protection options. ECC can either protect every byte with an additional 5-bit ECC code or every 32-bit word with an additional 7-bit ECC code. For instruction memories, ECC always protects every 32-bit word with an additional 7-bit ECC code. For all cache-tag arrays, each tag entry is protected by an additional 7-bit ECC code.

The Xtensa ECC option corrects single-bit errors in the data memories (data RAM and the data cache's data and tag arrays) in hardware on the fly through a hardware instruction replay, which allows program execution to continue while incurring only a single replay penalty.

When ECC on-the-fly error correction is enabled, it is important to note that the error is not actually corrected in the memory itself. Only the value delivered to the processor is corrected. A software scrubbing routine is required to periodically purge the memory of soft errors. (Hard errors cannot be purged this way.) For errors in the data RAMs, instruction RAMs, and the data cache, the software routine need only write the correct data back to memory. For errors in the instruction cache, the software should invalidate the cache line to force a cache-line reload.

When 7-bit ECC code is selected to protect 32 bit words on the Data memories, there are performance implications to consider. Because all data transactions must be performed at a 32-bit granularity, instructions that store less than 32 bits to the data memo-

ries will require a read modify write transaction to occur. Likewise, any store that uses the TIE `StoreByteDisable` interface will require a read modify write transaction as well. Because these store instructions perform a load as well as a store, a long sequence of these narrow store instructions may cause the store buffer to fill up prematurely which may cause additional stalls in the processor.

9.15 Processor Interface (PIF) Options

- No-PIF configuration
  The designer can eliminate the processor interface (PIF) using a configuration option to reduce gate count. A processor configured without a PIF must operate entirely out of local memory.

  Note: Because the processor's cache-line fills and castouts will use the PIF to communicate with system memory, a processor configured without a PIF cannot support and therefore cannot have cache memories.
- PIF Width Option
  The PIF can be configured to be 32, 64, or 128 bits wide. The PIF read and write data buses are the same width.
- PIF Critical Word First Option
  The PIF can be configured to do cache line read requests in Critical Word First fashion, such that the needed data is returned first on the PIF. This requires that the Early Restart option be selected.
- PIF Arbitrary Byte Enables Option
  The PIF can be configured to use arbitrary byte enables on single PIF width store transactions. This can reduce the number of PIF requests on store throughs in the case that the core generates stores with arbitrary byte enables.
- PIF Request Attribute Option
  The PIF can be configured to have an additional Request Attribute output for a PIF master (POReqAttribute[11:0]) and input for a PIF slave (PIReqAttribute[11:0]). This option is useful for implementing system level features, such as a second level cache.
- Inbound-PIF Request Option
  Supports inbound-PIF requests to instruction and data RAM so that external logic including other processors and DMA controllers can access the processor's local-RAM ports via the PIF.

  Note: Inbound-PIF requests cannot access a processor's local ROM and XLMI port.
- Inbound-PIF-Request Buffer Size
  The inbound-PIF request buffer size can be configured to hold 2, 4, 8, or 16 entries.
- Write-Buffer Entry Capacity
  The PIF's write buffer size can be configured to hold 1, 2, 4, 8, 16, or 32 entries.

- Prioritize Loads Before Stores Option
  Allows the processor to reorder load requests before store requests on the PIF to reduce the load-miss penalty and improve performance. The decision to bypass store requests is made by comparing the eight address bits above the cache index—if the processor configuration has a data cache—or log2(LoadStore width)—if the processor has no data cache—for store operations currently residing in the write buffer.
- Write Response Option
  The processor's PIF can be configured to count write responses for memory ordering or to aid in the use of synchronization primitives. When write responses are configured, 16 unique IDs are allocated to PIF write requests. As many as 16 PIF write requests can be outstanding at any time. Write Response must be configured if both Power Shut Off and AHP-lite/AXI bridge are configured.

  Note: If the inbound-PIF configuration option is selected, the PIF write-response configuration option causes the PIF to send write responses to external devices that issue inbound-PIF write requests.

9.16 AHB-Lite and AXI Bus Bridges

When the PIF option is selected, an optional bridge from PIF to AHB-Lite or AXI can also be configured. These bridges enable easy integration of Xtensa processors into existing AHB-Lite or AXI systems. The AHB-Lite and AXI interfaces can be clocked in one of three ways relative to the Xtensa processor:

- The processor and bus clocks are synchronous and the same frequency.
- The processor and bus clocks are synchronous, but the processor clock frequency is an integer multiple faster than the bus frequency. Multiples up to a 4 to 1 clock ratio are allowed.
- The processor and bus clocks are asynchronous. An asynchronous FIFO synchronizes PIF requests and responses.

See Chapter 26, Chapter 27, and Chapter 28 for more details about the asynchronous FIFO, AHB-Lite bridge, and AXI bridge.

9.17 TIE Module Options

The processor has the following TIE module option:

- TIE Arbitrary-Byte-Enable Option
  Allows TIE-based load/store instructions to initiate bus operations with arbitrary byte-enable patterns. Arbitrary byte enables are useful, for example, when storing unaligned data at the beginning and end of a data array.

9.18 Test and Debug Options

The processor has the following test and debug options:

- Scan Option
  Implements scan chains for SOC testability.
- Debug Option
  Implements instruction counting and breakpoint exceptions for debugging software or for other external-hardware uses. This option requires the high-priority interrupt option, requires the setting of a debug interrupt level, and requires the selection of the number of instruction breakpoint registers (0, 1, or 2) and the number of data breakpoint registers (0, 1, or 2). If 0 breakpoint registers are selected, the user will get only instruction counting and breakpoint exceptions.
- On-Chip Debug (OCD) Option
  Enables hardware access to software-visible processor state through the Access Port. This allows an external agent - such as a debugger running on a host PC - to connect to Xtensa and issue and execute instructions.

  This option also enables two new instructions LDDR32.P and SDDR32.P to speed up memory download/upload through the Debug Module. (Note that the Debug Instruction Register array option of RD and earlier releases is no longer available.)

  - External Debug Interrupt is a sub-option of OCD
    Adds break-in/break-out interfaces used to chain OCD interrupts of multiple processors.
- Trace Port Option
  Creates outputs that reflect the processor's internal state during normal operation. The option's data-trace sub-option parameter allows the trace port to report additional information about load/store values.
- TRAX Option

  Provides increased visibility into the activity of the processor core. Xtensa implements hardware that collects compressed PC trace data. This allows a user to non-intrusively examine the execution stream of the program running on Xtensa.

  There are three sub-options as follows.

  - Size of TraceRAM
  - Shared TraceRAM - allows multiple processors to share one physical RAM
  - ATB output - implements AMBA Trace Bus interface so that trace may be output to an ARM CoreSight™ system. Requires the APB option described below.
- The Performance Monitor option implements hardware used to count performance-specific events such as cache misses. A "profiling" type interrupt is also created.
  - The number of performance counters is a sub-option.

- APB option - Access from the external world to Debug and PSO functions can happen through an optional APB slave port on Xtensa. Access via this interface would be used in addition to access via the JTAG interface.

  When the APB option is selected, the Debug module also becomes compatible with the CoreSight debug standard available from ARM.

The above Test and Debug options are described in more detail in the *Xtensa Debug Guide*.

9.19 Power Management Options and Features

The processor has the following power management options and features:

- TIE Semantic Data Gating Option

  Allows the insertion of data gates on the inputs of TIE semantics to prevent unnecessary toggling on semantics not currently in use by the core. You can specify whether all semantics should be data gated, or only those with the user-defined TIE data gating attribute.

- Memory Data Gating Option

  Allows the insertion of data gates on the outbound address and data lines of both instruction and data memories, saving idle-cycle memory dynamic power.

- Power Shut Off Option

  Creates a power control module (PCM) and CPF/UPF files to partition the processor subsystem into separate power domains. Unused domains can then be powered down, saving leakage power. Software shutdown and wakeup routines are also provided. In addition, you can configure full state retention of the processor power domain, thereby providing fast wakeup without the power loss associated with saving state to memory. Power Shut Off is a feature-controlled portion of the product. To gain access to this functionality, contact your Cadence representative.

- WAITI

  Allows the processor to suspend operation until an interrupt occurs by executing the optional WAITI instruction.

  Note: WAITI is available only when one or more interrupts have been configured.

- Global Clock-Gating Option

  Allows the processor generator to add clock gating to the main processor clock, which is routed to large portions of the processor's internal logic. This option can be used in conjunction with the functional clock-gating option.

- Functional Clock-Gating Option

Allows the processor generator to add clock gating to the processor's registers so that register clocking is based on register usage throughout the processor. This option can be used in conjunction with the global clock-gating option.

- External Run/Stall Control Signal

  This processor input allows external logic to stall large portions of the processor's pipeline by shutting off the clock to much of the processor's logic to reduce operating power when the processor's computational capabilities are not immediately needed by the system.

  Note: Using the `WAITI` instruction to power down the processor will save more power than use of the external run/stall signal because the `WAITI` instruction disables more of the processor's internal clocks.

- Instruction Fetch Width Option

  A wider instruction-fetch width (64- or 128-bit) may be selected to reduce the number of memory accesses by allowing more instructions to be fetched in one cycle. This option reduces instruction-memory power dissipation by fitting multiple narrow instructions into the processor's holding buffer. However, these power savings may not be realized with branch-intensive code because many instructions may be fetched but not executed. Power and code analysis can help to decide on this option setting.

  Notes:

  - The instruction fetch width must be set to: 32 or 64 bits if no wide instructions are configured or if 32 bits is the maximum instruction size; 64 bits if the maximum instruction size is between 40 bits and 64 bits; or 128 bits if the maximum instruction size is between 72 bits and 128 bits.
  - Instruction-RAM and instruction-ROM port widths are always equal to the instruction-fetch width but instruction cache width may be greater than or equal to the instruction-fetch width. For minimum power, instruction-cache width should be equal to instruction-fetch width.

- L0 Loop Buffer Option
  If the Zero Overhead Loop Option is selected, an optional L0 Loop Buffer is added that will capture instructions from the body of a Zero Overhead Loop, and then execute subsequent iterations of the loop from this buffer. This allows the instruction memories to not be enabled during most of the loop execution. Power savings will depend on what portion of code is being executed as Zero Overhead Loops.

  - L0 Loop Buffer Size
    The L0 Loop Buffer Size can be 16 Bytes to 256 Bytes in powers of two, but must be a minimum of four Instruction Fetch Widths.

9.20 Incompatible Configuration Options

This section lists the processor configuration options that are not compatible with each other.

Asymmetric PIF, Data Memory/Cache, Instruction Fetch, Instruction Cache widths are allowed. The following table shows the allowed width configurations.

Table 9–14. Memory and PIF Width Options

PIF Width (bits)	Local Data Memory Width (bits)	32-bit Instruction Fetch 32-bit IRAM, IROM (if any)			64-bit Instruction Fetch 64-bit IRAM, IROM (if any)			128-bit Instruction Fetch 128-bit IRAM, IROM (if any)		
		32-bit Instr Cache	64-bit Instr Cache	128-bit Instr Cache	32-bit Instr Cache	64-bit Instr Cache	128-bit Instr Cache	32-bit Instr Cache	64-bit Instr Cache	128-bit Instr Cache
32	32	Yes	No	No	No	Yes	No	No	No	Yes
32	64	Yes	No	No	No	Yes	No	No	No	Yes
32	128	Yes	No	No	No	Yes	No	No	No	Yes
32	256	Yes	No	No	No	Yes	No	No	No	Yes
32	512	Yes	No	No	No	Yes	No	No	No	Yes
64	32	No	No	No	No	No	No	No	No	No
64	64	Yes	Yes	No	No	Yes	No	No	No	Yes
64	128	Yes	Yes	No	No	Yes	No	No	No	Yes
64	256	Yes	Yes	No	No	Yes	No	No	No	Yes
64	512	Yes	Yes	No	No	Yes	No	No	No	Yes
128	32	No	No	No	No	No	No	No	No	No
128	64	No	No	No	No	No	No	No	No	No
128	128	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes
128	256	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes
128	512	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes

- Further memory-width restrictions are as follows:
  - All data-memory/cache ports accessed by the load/store units have the same width, that is, data RAM, data ROM, data cache, XLMI have the same width.
  - All data-memory/cache port widths must be greater than or equal to PIF width.
  - Instruction RAM and instruction ROM must have the same width as the instruction-fetch width (32, 64, or 128 bits), which must be greater than or equal to maximum instruction size (24, 32, 64, or 128 bits).
  - Instruction-cache width must be greater than or equal to the instruction-fetch width (32, 64, or 128 bits) and it must be less than or equal to the maximum of the instruction-fetch or PIF width.

- For a no PIF option with a small 4K instruction RAM, restrictions are as follows:
  - No relocatable vectors
  - Call0 ABI only (Windowed Reg Files not allowed)
  - No ECC / Parity
  - For high level interrupts enabled, the max int level is 3. NMI must be placed at level 3
- MMU with TLB and Autorefill is incompatible with the following options:
  - Multiple instruction RAM's
  - Memory parity and ECC
  - Multiple load/store units
  - Prefetch to L1 Data Cache Option
  - Block Prefetch Data Cache Option
- MPU requires the following features to be configured:
  - iDMA
  - Exclusive Store
  - SuperGather
  - ACE_Lite
- MPU has the following restrictions:
  - D$: Write-Back only
  - I$, D$: if either is configured, require the Dynamic way disable option
  - AXI4 required
- The following options must NOT be configured along with the MPU:
  - Region Protection Unit
  - Region Protection With Translation MMU
  - MMU with Translation Look Aside Buffer (TLB) and Autorefill
  - Hardware unaligned handling
  - XLMI
  - Conditional store synchronization (S32C1I)
- The iDMA requires the following options to be configured:
  - AXI4
  - MPU
  - LIttle endian
  - iDMADone and iDMAErr interrupts

- Exclusive access has the following restrictions:
  - MPU is required
  - AXI4 is required
  - XLMI must not be configured
  - Write response is required
- Bus ECC/Security requires the following:
  - Bus ECC and security must be both selected or neither selected
  - AXI4
  - OCD (debug)
  - Write response
- Memory ECC/Parity Support configurations have the following restrictions:
  - All instruction-fetch interfaces (instruction cache, instruction RAM) configured with memory-error protection must have the same type of Memory ECC/Parity support. Some or all of these interfaces can be configured without memory-error protection.
  - All data-memory/cache interfaces (data cache, data RAM) configured with memory-error protection must have the same type of Memory ECC/Parity support. Some or all of these interfaces can be configured without memory-error protection.
  - Memory errors are not supported for processors configured with unaligned load handled by hardware and two load/store units with caches
  - In order to configure Prefetch Memory Errors (parity or ECC), at least one other local memory or cache (Instruction or Data) must have memory error checking configured.
- In configurations with Write-Back Cache, Write-Buffer Entry capacity must be greater than or equal to (data cache Line Size * 8 / Write-Buffer width), where Write-Buffer width is equal to the data-memory/cache port width.

- The XLMI Port has the following configuration limitations:
  - Unaligned Load/Store action is handled by hardware
  - No Conditional Store Synchronization Instruction
  - No Connection Box (C-Box)
  - No Memory ECC/Parity Support
  - No Dual load/store units
  - No Inbound PIF
  - Multiple banks not supported
  - Data memory port width is up to 128 bits
  - No MPU--no exclusive access, iDMA, ACE_Lite, and Bus ECC
  - Not compatible with the following DSP options: Vision P5, Vision P6, and Fusion G3
- ACE_Lite has the following requirements:
  - AXI4
  - MPU
  - Write response
  - No Conditional store (S32C1I)
- The DSP option only supports little endian configuration--big endian configuration is not supported
- SuperGather (Gather/Scatter) option is only for Vision P5 and Vision P6, and is required
- The Fusion G3 DSP has the following restrictions:
  - Requires PIF, AHB_lite, AXI3, or AXI4
  - No PIF is not supported

9.21 Special Register Reset Values

The Xtensa Instruction Set Architecture (ISA) defines the reset value of all architectural special registers. The majority of the special registers have an undefined reset value, and each implementation may choose a different reset value. For greatest portability, software should be written in a manner that does not assume the reset value of a given implementation, but rather only relies on the architectural definition.

The following table defines the reset values of special registers in the Xtensa LX7 Processor implementation. The registers that are specified as "undefined" by the Xtensa ISA may have a different reset value in other implementations.

Table 9–15. Special Register Reset Values

Register Name	Architectural Reset State	XtensaLX7 Reset State
ACCHI	Undefined	0
ACCLO	Undefined	0
ATOMCTL	0x28	0x28
CACHEADRDIS	0x0	0x0
CCOMPARE0..2	Undefined	0
CCOUNT	Undefined	0
CPENABLE	Undefined	1<<CoprocCount - 1
DBREAKA0..2	Undefined	0
DBREAKC0..2	Undefined	0
DEBUGCAUSE	Undefined	0
DDR	Undefined	0
DEPC	Undefined	0
DTLBCFG	0	0
EPC1	Undefined	0
EPC2..7	Undefined	0
EPS2..7	Undefined	0
ERACCESS	0x0	0x0
EXCCAUSE	Undefined	0
EXCSAVE1	Undefined	0
EXCSAVE2..7	Undefined	0
EXCVADDR	Undefined	0
FCR	Undefined	0
FSR	Undefined	0
GSERR	Undefined	0
GSMEA	Undefined	0
GSMES	Undefined	0
IBREAKA0..2	Undefined	0
IBREAKENABLE	$0^{NIBREAK}$	0
ICOUNT	Undefined	0
ICOUNTLEVEL	Undefined	0
INTERRUPT	Undefined	0
INTENABLE	Undefined	0
ITLBCFG	0	0

Table 9–15. Special Register Reset Values (continued)

Register Name	Architectural Reset State	XtensaLX7 Reset State
LBEG	Undefined	0
LCOUNT	Undefined	0
LEND	Undefined	0
LITBASE	Bit-0 clear, others undefined	0
M0..3	Undefined	0
MECR	Undefined	0
MEMCTL	0x0 or 0x1	If LoopBufferSize>0 then 0x1, else 0x0
MEPC	Undefined	0
MEPS	Undefined	0
MESAVE	Undefined	0
MESR	0xXXXX0C00	0xC00
MEVADDR	Undefined	0
MISC0..3	Undefined	0
MMID	Undefined	
MPUCFG	Number of foreground segments	Number of foreground segments
MPUENB	0x0	0x0
PERFCTRL	Undefined	0
PREFCTL	0	0
PRID	Set to value on input pins	Set to value on input pins
PS	If Interrupt option enabled then 0x1F, else 0x10	0x1F or 0x10
PTEVADDR	Undefined	0
RASID	0x04030201	0x4030201
SAR	Undefined	0
SCOMPARE1	Undefined	0
THREADPTR	Undefined	0
VECBASE	Undefined	Config dependent
WINDOWBASE	Undefined	0
WINDOWSTART	Undefined	0x1

10. Xtensa LX7 Bus and Signal Descriptions

Figure 10–27 shows the major ports going into and out of the Xtensa LX7 processor. Ports that are actually present, as well as the width of many of the buses, depend on the specific configuration.

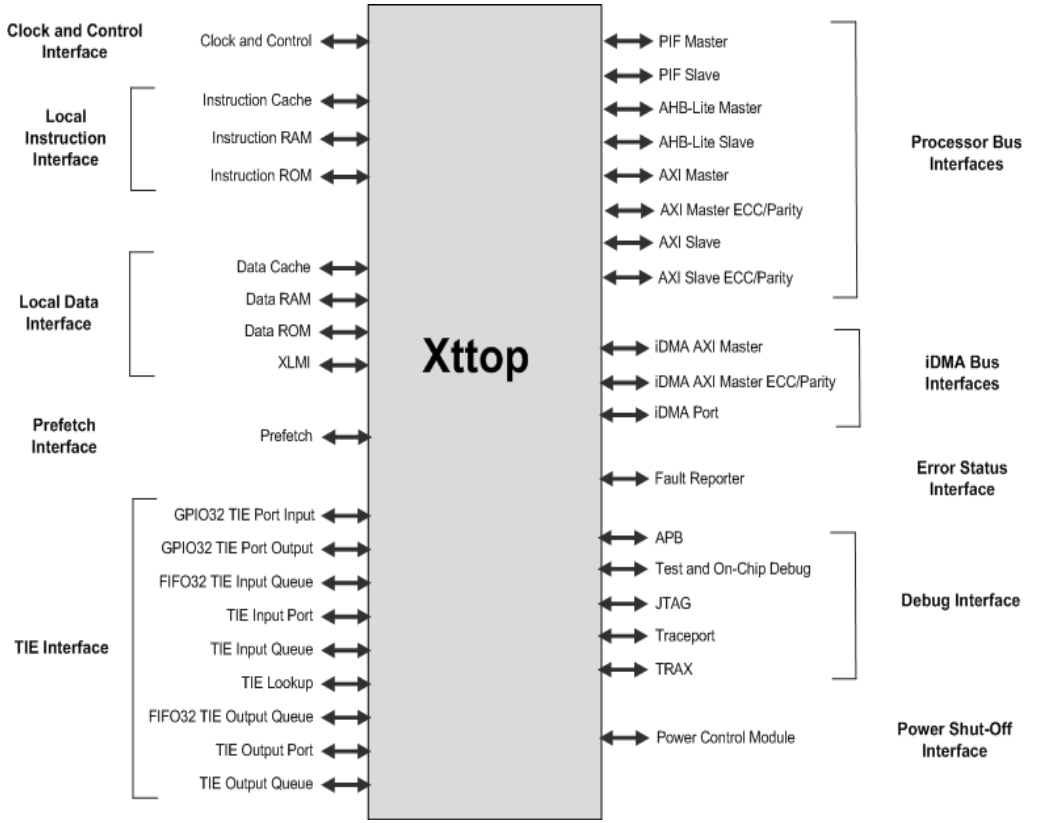


Figure 10–27. Xttop Block Diagram Showing Major Interfaces

The top level of design hierarchy of the Xtensa processor is named Xttop, as described in Section 8.1 “Design Hierarchy” on page 91. Ports at the Xttop level of the hierarchy are described in the tables below.

Note that signals to/from memories – i.e., data RAM, Traceport, etc. go between Xttop and the memory concerned. In other words they are not inputs or outputs of the Xtmem hierarchy. With regard to other signals, they are inputs or outputs of Xtmem, but simply passed through to the Xttop hierarchy. Exceptions are expressly noted.

10.1 Clock and Control Interface

Table 10–16 lists the clock and control interface signals.

Table 10–16. Clock and Control Interface Signals

Name (Configured Width)	I/O	Description	Notes
CLK	Input	Clock input to the processor. Note: CLK is used to generate the internal processor clock signal PCLK	Except for the clock gating logic that uses latches, the Xtensa design uses all edge-triggered flip-flops. Given that fact, the Xtensa IP does not impose any duty cycle requirements on any of its clocks. The implementation scripts in the hardware package specifies a 50% duty cycle, but you can choose any duty cycle that meets timing in your sign-off flow.
BCLK	Input	Bus clock when core and bus clocks are asynchronous	Requires: Asynchronous AMBA bus option and either AHB-Lite or AXI bridge
Strobe	Input	Indicates phase of the bus clock to support synchronous clock ratios between the core and bus	Requires: AHB-Lite or AXI bridge, and not asynchronous AMBA bus
BReset	Input	Reset input to the processor. BReset is active high and is synchronized inside of the Xttop module	Both synchronous and asynchronous flip-flop implementation options are supported.
PReset	Output	BReset synchronized to the processor domain clock	Used in 7-stage pipeline configurations to reset memory data staging registers. Not used in 5-stage pipelines.
PWaitMode	Output	Indicates that the processor is in sleep mode. The processor asserts this signal when it has executed a WAITI instruction and is waiting for an interrupt. Any asserted interrupt that is not disabled will wake the processor, which will then jump to the appropriate designer-configured interrupt vector	
RunStall	Input	This external signal stalls the processor. This signal is registered within the processor before it is used	The stall condition is indicated with a logic high assertion.
PRID[15:0]	Input	Input to the processor, latched at reset into the low-order bits of the PRID (processor ID) special register	Requires: PRID

Table 10–16. Clock and Control Interface Signals (continued)

Name (Configured Width)	I/O	Description	Notes
StatVectorSel	Input	Selects between one of two stationary vector bases (0: default, 1: alternative). The reset vector and memory error exception handler vector addresses are fixed offsets from the stationary vector base. The two stationary vector bases are configurable. The StatVectorSel pin must be held stable for at least 10 CLK cycles before the falling edge of BReset, and for 10 CLK cycles after the falling edge of BReset. See Section 22.3.4 “Full Reset of an Asynchronous-Reset Xtensa”, Section 22.3.5 “Full Reset of a Synchronous-Reset Xtensa”, and Section 22.4 “Static Vector Select” for more information.	Requires: Relocatable Vectors

Table 10–16. Clock and Control Interface Signals (continued)

Name (Configured Width)	I/O	Description	Notes
AltResetVec[31:0]	Input	When the Relocatable Vectors option is configured, then the sub-option External Reset Vector can be selected. When selected, these pins provide an alternate reset vector address. The timing of these signals is exactly the same as StatVectorSel, above. For more information, see Section 22.4.1 “External Reset Vector Sub-Option”.	Requires: Relocatable Vectors AND External Reset Vector
BInterrupt [number of interrupts-1:0]	Input	External processor interrupts (maximum 32) can be configured. Each interrupt input can be configured to be level- or edge-triggered. If the processor is not busy, and the interrupt source is not software masked (except NMI interrupt), assertion of an interrupt input causes code execution to jump to the appropriate interrupt handler routine at a specified vector location. One of the interrupt bits can be configured by the designer to be a non-maskable interrupt (NMI) Note: Certain conditions can delay interrupts	Requires: Interrupts
ErrorCorrected	Output	Indicates that a correctable memory error has been corrected by the hardware	Requires: Data Ram, Data Cache, or Instruction RAM ECC

10.2 Local Instruction Interfaces

This section lists the instruction cache, instruction RAM, and instruction ROM interface signals.

Table 10–17. Instruction Cache Interface Port Signals

Name	I/O	Description	Notes
ICache<*way*>Data [icwidth-1:0][5]	Input	Input data from instruction cache array (per way)	
ICache<*way*>CheckData [iccheckwidth-1:0][7]	Input	Error check bits from instruction cache array (per way)	Requires ICache parity or ECC
ICache<*way*>En	Output	Instruction cache array enable (per way)	
ICache<*way*>WordEn [icwordwidth-1:0][9]	Output	Instruction cache array word enables (per way).	Requires ICache width > 32 bits
ICache<*way*>Wr	Output	Instruction cache write enable (per way) — valid during refills and special cache instructions	
ICacheAddr [icsetawidth+icwordoffset-1: icwordoffset][2,3]	Output	Instruction cache array address lines	
ICacheWrData [icwidth-1:0][5]	Output	Output data to be written to instruction cache array	
ICacheCheckWrData [iccheckwidth-1:0][7]	Output	Error check bits to be written to instruction cache array	Requires ICache parity or ECC
ITag<*way*>Data [itwidth-1:0][4]	Input	Data read from instruction cache tag array (per way)	
ITag<*way*>CheckData [itcheckwidth-1:0][8]	Input	Error check bits from instruction cache tag array (per way)	Requires ICache parity or ECC
ITag<*way*>En	Output	Instruction cache tag array enable (per way)	
ITag<*way*>Wr	Output	Instruction cache tag array write enable (per way)	

Table 10–17. Instruction Cache Interface Port Signals (continued)

Name	I/O	Description	Notes
ITagAddr [icsetbyteaddrwidth-1: iclineoffset][1,6]	Output	Instruction cache tag array address lines	
ITagWrData [itwidth-1:0][4]	Output	Output data to be written to instruction cache tag array	
ITagCheckWrData [itcheckwidth-1:0][8]	Output	Error check bits to be written to instruction cache tag array	Requires ICache parity or ECC

Notes:
1. icsetbyteaddrwidth =(log2(ICache size [in bytes]/ways)
2. icsetawidth = (log2((Icache size/icbytes)/ways)
3. icwordoffset = (log2(icwidth/8)
4. itwidth = (ITag access width [in bits])
5. icwidth = (ICache access width [in bits])
6. iclineoffset = (log2(ICache line size [in bytes])8
7. iccheckwidth = (icwidth/32) for parity, (icwidth*7)/32 for ECC
8. itcheckwidth = 1 for parity, 7 for ECC
9. icwordwidth = (icwidth/32))

Table 10–18. Instruction RAM Interface Port Signals

Name (Configured width)	I/O	Description	Notes
IRam*n*Addr [irambyteaddrwidth-1:isize][1,2,3]	Output	Instruction RAM address lines.	
IRam*n*Busy[1]	Input	Instruction RAM memory unavailable during the previous cycle. Note: There are combinational paths from IRam*n*-Busy to all memory address, data, and control outputs.	Requires IRamBusy
IRam*n*Data [iramwidth-1:0][1,4]	Input	Input data lines from instruction RAM.	
IRam*n*CheckData [iramcwidth-1:0][1,5]	Input	Error check bits from instruction RAM.	Requires IRAM parity or IRAM ECC
IRam*n*En[1]	Output	Instruction RAM enable.	

Table 10–18. Instruction RAM Interface Port Signals (continued)

Name (Configured width)	I/O	Description	Notes
IRam*n*WordEn [iramwordwidth-1:0][1,6]	Output	Word enables for 64-bit or 128-bit wide instruction RAMs. Instruction RAM loads and stores are always 32 bits. (See Section 18.3.1).	Requires 64-bit or 128-bit wide instruction fetch
IRam*n*LoadStore[1]	Output	Load or store transaction (as opposed to an instruction fetch) is directed at instruction RAM. (See Section 18.3.1).	
IRam*n*Wr[1]	Output	Instruction RAM write line.	
IRam*n*WrData [iramwidth-1:0][1,4]	Output	Data to be written to instruction RAM.	
IRam*n*CheckWrData [iramcwidth-1:0][1,5]	Output	Error check bits to be written to instruction RAM.	Requires IRAM parity or IRAM ECC

Notes:

1. n is the number of the associated instruction RAM (*n* = 0 if there is one RAM and *n* = {0, 1} if there are two RAMs.)

2. isize = 3 if 64-bit instruction fetch is configured, 2 otherwise

3. irambyteaddrwidth = log2(IRam size [in bytes])

4. iramwidth = IRam access width [in bits]

5. iramcwidth =(iramwidth/32) for parity, (iramwidth*7/32) for ECC

6. iramwordwidth=(iramwidth/32)

Table 10–19. Instruction ROM Interface Port Signals

Name (Configured Width)	I/O	Description	Notes
IRom0Addr [irombyteaddrwidth-1:isize][1,2]	Output	Instruction ROM address lines.	
IRom0Busy	Input	Instruction ROM memory unavailable during the previous cycle. Note: There are combinational paths from IRom*n*Busy to all memory address, data, and control outputs.	Requires IRomBusy
IRom0Data[iromwidth-1:0][3]	Input	Input data lines from instruction ROM.	

Table 10–19. Instruction ROM Interface Port Signals (continued)

Name (Configured Width)	I/O	Description	Notes
IRom0En	Output	Instruction ROM enable.	
IRom0WordEn [iromwordwidth-1:0][4]	Output	Word enables for 64-bit or 128-bit wide instruction ROMs. Instruction ROM loads are always 32 bits. (See Section 21.1.).	Requires 64-bit or 128-bit wide instructions
IRom0Load	Output	Load is occurring from instruction ROM. (See Section 21.1.).	

Notes:

1. isize = 3 if 64-bit FLIX instructions are configured, 2 otherwise
2. irombyteaddrwidth = log2(IRom size [in bytes])
3. iromwidth = IRom access width [in bits]
4. iromwordwidth = (iromwidth/32)

10.3 Local Data Interfaces

This section lists the data cache, data RAM, data ROM, and XLMI interface signals.

Table 10–20. Data Cache Interface Port Signals

Name (Configured Width)	I/O	Description	Notes
DCache<*way*>Data<*n*> [dcwidth-1:0][3,9]	Input	Input data from data cache array (per way).	
DCache<*way*>CheckData<*n*> [dccheckwidth-1:0][7,9]	Input	Error check bits from data cache array (per way).	Requires DCache Parity or ECC
DCache<*way*>En<*n*>[9]	Output	Data cache array enable (per way).	
DCache<*way*>Wr<*n*>[9]	Output	Data cache write-enable (per way).	
DCacheAddr<*n*> [dcsetbyteaddrwidth-1: dcaccessoffset][1,5,9]	Output	Data cache array address lines.	
DCacheByteEn<*n*> [dcbytes-1:0][4, 9]	Output	Data cache byte enables.	
DCacheWrData<*n*> [dcwidth-1:0][3, 9]	Output	Output data to be written to data cache array.	
DCacheCheckWrData<*n*> [dccheckwidth-1:0][7, 9]	Output	Error check bits to be written to data cache array.	Requires DCache Parity or ECC
DTag<*way*>Data<*m*> [dtwidth-1:0][2, 10]	Input	Input data read from data cache tag array (per way).	
DTag<*way*>CheckData<*m*> [dtcheckwidth-1:0][8, 10]	Input	Error check bits from data cache tag array (per way)	Requires DCache Parity or ECC
DTag<*way*>En<*m*>[10]	Output	Data cache tag array enable (per way).	
DTag<*way*>Wr<*m*>[10]	Output	Data cache tag array write enable (per way).	

Table 10–20. Data Cache Interface Port Signals (continued)

DTagAddr<*m*> [dcsetbyteaddrwidth-1: dclineoffset][1,6, 10]	Output	Data cache tag array address lines.	
DTagWrData<*m*> [dtwidth-1:0][2]	Output	Data to be written to data cache tag array.	
DTagCheckWrData<*m*> [dtcheckwidth-1:0][8, 10]	Output	Error check bits to be written to data tag array.	Requires DCache Parity or ECC

Notes:

1. dcsetbyteaddrwidth= log2 (DCache size [in bytes]/ways)
2. dtwidth = DTag access width [in bits]
3. dcwidth = DCache access width [in bits]
4. dcbytes = dcwidth / 8
5. dcaccessoffset = log2(dcbytes)
6. dclineoffset = log2(DCache line size[in bytes])
7. dccheckwidth = (dcwidth/8) for Byte Parity, (dcwidth*5/8) for Byte ECC (dcwidth/32) for Word Parity, (dcwidth*7/32) for Word ECC
8. dtcheckwidth = 1 bit for Parity, 7 bits for ECC
9. n = if number of loadstore units > 1, then n = B<bank#> else n = loadstore unit number
10. *m* = ls unit number or 'Aux' if Prefetch to L1 or if Coherence option is configured

Table 10–21. Data RAM Interface Port Signals

Name (Configured Width)	I/O	Description	Notes
DRam*n*Addr*m* [drambyteaddrwidth-1:dramaccessoffset][1,2,3,6]	Output	Data RAM address lines.	
DRam*n*ByteEn*m* [drambytes-1:0][1,2,5]	Output	Data RAM byte enables.	
DRam*n*Data*m* [dramwidth-1:0][1,2,4]	Input	Input data lines from data RAM.	
DRam*n*CheckData*m* [dramcheckwidth-1:0][1,2,7]	Input	Error check bits from data RAM.	Requires data RAMParity or ECC
DRam*n*En*m*[1,2]	Output	Data RAM enable.	
DRam*n*Busy*m*[1,2]	Input	Data RAM memory unavailable during the previous cycle. Note: There are combinational paths from DRam*n*Busy to all memory address, data, and control outputs.	Requires DRamBusy

DRam*n*Wr*m*[1,2]	Output	Data RAM write line.	
DRam*n*WrData*m* [dramwidth-1:0][1,2,4]	Output	Data to be written to data RAM.	
DRam*n*CheckWrData*m* [dramcheckwidth-1:0][1,2,7]	Output	Error check bits to be written to data RAM.	Requires data RAM Parity or ECC
DRam*n*Lock*m*[1,2]	Output	Indicates that an atomic transaction (e.g. S32C1I or RCW) has locked the data RAM interface.	Requires Conditional Store Synchronization

Notes:

1. *n* is the number of the associated data RAM (*n*=0 if the configuration has one data RAM. *n*={0, 1} if the configuration has two data RAMs.)
2. *m* is {B0,B1,B2,B3} for multiple-bank data RAMs (up to 4 banks); or is {0/1} indicating associated load/store unit; or is blank if C-Box is configured for single-bank data RAMs.
3. drambyteaddrwidth = $\log_2$(data RAM size [in bytes])
4. dramwidth = data RAM access width [in bits]
5. drambytes = dramwidth / 8
6. dramaccessoffset = $\log_2$(drambytes) + $\log_2$(data RAM banks)
7. dramcheckwidth = (drambytes) for Parity, (dramwidth*5/8) for ECC

Note: An Xtensa core configured with two load/store units can be configured to open its load/store interfaces to data RAM/ROM for an external agent. If configured, additional ports will be generated. For details, refer to Section 18.10 “Exposed Processor Interface for a Customer-Designed C-Box”.

When the SuperGather engine is configured, the data RAM interface is further partitioned into sub-banks to improve memory performance. The physical parameters of a data RAM sub-bank are determined jointly by the data RAM parameters and the gather/scatter element width. Their relationship is summarized as follows:

- drambyteaddrwidth = $\log_2$(data RAM size [in bytes])
- dramwidth = data RAM access width [in bits]
- drambytes = dramwidth / 8
- dramaccessoffset = $\log_2$(drambytes) + $\log_2$(data RAM banks)
- subbankwidth = gather/scatter element width in bit
- subbankcount = dramwidth/subbankwidth
- subbankbytes = subbankwidth / 8

- subbankcheckwidth: number of check bits per subbankwidth,as follows:
  - = 1x(subbankbytes) for byte-level parity protection;
  - = 5x(subbankbytes) for byte-level ECC protection;
  - = 1x(subbankbytes/4) for word-level parity protection;
  - = 7x(subbankbytes/4) for word-level ECC protection.

Using the above parameters, the data RAM sub-bank interface is described in Table 10–22.

Table 10–22. Data RAM Sub-bank Interface Port Signals

Name (Configured Width)	I/O	Description	Notes
DRam*n*AddrB*m*S*u* [drambyteaddrwidth-1:dram-accessoffset][1,2,3]	Output	Address lines for each data RAM sub-bank.	
DRam*n*ByteEnB*m*S*u* [subbankbytes-1:0][1,2,3]	Output	Byte enables for each data RAM sub-bank.	
DRam*n*DataB*m*S*u* [subbankwidth-1:0][1,2,3]	Input	Input data lines from each data RAM sub-bank.	
DRam*n*CheckDataB*m*S*u* [subbankcheckwidth-1:0][1,2,3]	Input	Read check bits for each data RAM sub-bank.	Requires data RAM Parity or ECC
DRam*n*EnB*m*S*u*[1,2,3]	Output	Data RAM enable for each sub-bank.	
DRam*n*WrB*m*S*u*[1,2,3]	Output	Data RAM write line for each sub-bank.	
DRam*n*WrDataB*m*S*u* [subbankwidth-1:0][1,2,3]	Output	Write data for each data RAM sub-bank.	
DRam*n*CheckWrDataB*m*S*u* [subbankcheckwidth-1:0][1,2,3]	Output	Write check bits for each data RAM sub-bank.	Requires data RAM Parity or ECC
DRam*n*BusyB*m*[1,2,3]	Input	Busy signal for each data RAM bank.	Requires DRAM-Busy
DRam*n*LockB*m*[1,2,3]	Output	Lock signal for each data RAM bank.	Requires Conditional Store Synchronization

Notes:

1. n is the number of the associated data RAM (*n*=0 if the configuration has one data RAM. *n*={0, 1} if the configuration has two data RAMs.)

2. m is {0,1,2,3} for multiple-bank data RAMs (up to 4 banks);.

3. *u* is the sub-bank suffix (0..subbankcount-1)

Note: The optional Busy and Lock signals are instantiated per bank. All sub-banks that belong to the same bank share a single Busy and a single Lock when configured.

Table 10–23. Data ROM Interface Port Signals

Name (Configured Width)	I/O	Description	Notes
DRom0Addr*m* [drombyteaddrwidth-1:dromaccessoffset][1,2,4]	Output	Data ROM address lines.	
DRom0Data*m* [dromwidth-1:0][1,3]	Input	Input data from data ROM.	
DRom0En*m*[1]	Output	Data ROM enable.	
DRom0Busy*m*[1]	Input	Data ROM memory unavailable during previous cycle. Note: There are combinational paths from DRom*n*Busy to all memory address, data, and control outputs.	Requires DRomBusy
DRom0ByteEn*m*[1]	Output	Data ROM byte enables.	

Notes:

1. *m* is {0/1} if the associated load/store unit is directly connected to data ROM; or is blank if the C-Box is configured for single-bank data ROMs; or is {B0,B1,B2,B3} if the C-Box is configured for multiple-bank data ROMs (up to 4 banks)

2. drombyteaddrwidth = log2(data ROM size [in bytes])

3. dromwidth = data ROM access width [in bits]

4. dromaccessoffset = log2(dromwidth / 8) + log2(data ROM banks)

Table 10–24. XLMI (Xtensa Local Memory Interface) Port Signals

Name (Configuration Width)	I/O	Assertion Timing	Description	Notes
DPort0En*m*[1]	Output	N_{En}[5]	XLMI port 0 enable.	
DPort0Addr*m* [awidth-1:0][2]	Output	N_{En}[5]	Virtual address to XLMI port - only bits [awidth-1:dportaccessoffset] are used	
DPort0ByteEn*m* [dportbytes-1:0] [1,4]	Output	N_{En}[5]	XLMI port 0 byte enables.	
DPort0Wr*m*[1]	Output	N_{En}[5]	XLMI port 0 write enable.	
DPort0WrData*m* [dportwidth-1:0][1,3]	Output	N_{En}[5]	Data to be written to XLMI port 0.	
DPort0Busy*m*[2]	Input	$N_{En} + 1$[5]	Load or Store transaction not accepted by XLMI port 0 (present only if Busy option selected). Note: There are combinational paths from DPort0Busy*m* to all memory address, data, and control outputs.	Requires XLMI Port Busy
DPort0Load*m*[2]	Output	$N_{En} + 1$[5]	Load transaction to XLMI port 0 has begun.	
DPort0Data*m* [dportwidth-1:0][2,4]	Input	$N_{En} + N_{Access}$[5,6]	Input data from XLMI port 0.	
DPort0LoadRetired*m*[2]	Output	$N_{En} + N_{Access} + N_{Stall}$[5,6,7]	Oldest outstanding load from XLMI port 0 committed.	
DPort0RetireFlush*m*[2]	Output	$N_{En} + N_{Access} + N_{Stall}$[5,6,7]	All outstanding loads from XLMI port 0 will not commit.	

Notes:

1. m = number of the associated load/store unit. (0 for the Xtensa processor.)

2. awidth = virtual address width, always 32-bits

3. dportwidth = data access width of DPort0

4. dportbytes = dportwidth/8

5. N_{En} = cycle that DPort*n*En*m* is asserted

6. N_{Access} = memory access latency, i.e., 1 for a 5-stage pipeline or 2 for a 7-stage pipeline.

7. N_{Stall} = arbitrary number of stall cycles

10.4 Prefetch Interface

Table 10–25 lists the Prefetch port signals.

Table 10–25. Prefetch Port Signals

Name	I/O	Description	Notes
PrefetchRamAddr [prefetchaddrwidth-1:0][3]	Output	Prefetch RAM Address	
PrefetchRamEn	Output	Prefetch RAM Enable	
PrefetchRamPIFWEn[prefetch datawidth/pifwidth-1:0][1]	Output	Prefetch RAM PIF Width Enable	
PrefetchRamWr	Output	Prefetch RAM write operation	
PrefetchRamData [prefetchdatawidth-1:0][2]	Input	Prefetch RAM Read Data	
PrefetchRamCheckData [prefetchcheckwidth-1:0][4]	Input	ECC check or Parity bits for the PrefetchRamData	Requires ECC option
PrefetchRamWrData [prefetchdatawidth-1:0] [2]	Output	Prefetch RAM Write Data	
PrefetchRamCheckWrData [prefetchcheckwidth-1:0][4]	Output	ECC Check or Parity bits for the PrefetchRamWrData	Requires ECC option
PrefetchRamErrorUncorrected	Output	A double bit error has been detected on a Prefetch RAM read. It was uncorrectable.	Requires Prefetch Memory Errors to be configured
PrefetchRamErrorCorrected	Output	A single bit error has been detected on the Prefetch RAM and has been corrected.	Requires Prefetch Memory ECC Errors to be configured

Notes:

1. pifwidth = PIF width in Bits. Write of each PIF width of the prefetch RAM is controlled separately.
2. prefetchdatawidth = If no early restart is configured this parameter is equal to the Cache width in Bits, if early restart is configured then this parameter is max (Cache Width in Bits, 2 * pifwidth).
3. prefetchaddwidth = log2(8*Cache line size in Bytes/(prefetchdatawidth/8)).
4. prefetchcheckwidth = ECC ? 7 * pifwidth/32 : Parity ? pifwidth/32.

See Section 17.2.2 for details of when Prefetch applies to the instruction cache.

10.5 TIE Interfaces

This section lists the TIE GPIO32, FIFO32, input, output, and lookup signal interfaces.

Table 10–26. GPIO32 TIE Port Input (GPIO32 Option)

Name (Configured Width)	I/O	Description
TIE_IMPWIRE[31:0]	Input	A data value that is available for use via TIE instructions

Table 10–27. GPIO32 TIE Port Output (GPIO32 Option)

Name (Configured Width)	I/O	Description
TIE_EXPSTATE[31:0]	Output	Exported state data bus. The architectural copy of the state is exported on these wires

Table 10–28. FIFO32 TIE Input Queue Signals (QIF32 Option)

Name (Configured Width)	I/O	Description
TIE_IPQ[31:0]	Input	Input data bus connected to the output of the external queue. The data is registered in the processor before it is used by any instruction semantics
TIE_IPQ_PopReq	Output	Indicates that the processor is ready to read data from the input queue. This request is made independently of the Empty signal. The attached queue must ignore this signal if the queue is empty
TIE_IPQ_Empty	Input	Indicates the external queue is empty and cannot be popped

Table 10–29. TIE Input Port (per Designer-Defined Import Wire)

Name (Configured Width)	I/O	Description
TIE_<name> [width-1:0]	Input	Designer defined input-port. Import wires are automatically registered in the core and are available for use in the E-stage of an instruction.
Notes: There is one interface per import wire. TIE code: import_wire <name> <width>		

Table 10–30. TIE Input Queue Signals (per Designer-Defined Input Queue)

Name (Configured Width)	I/O	Description
TIE_<name> [iqwidth-1:0]	Input	Input data bus (or wire if 1-bit width) connected to the output of the external queue. The data is registered in the processor before it is used.
TIE_<name>_PopReq	Output	Indicates that the processor is ready to read data from the input queue. This request is made independent of the status of the empty signal and the attached queue is expected to ignore this signal if the queue is empty.
TIE_<name>_Empty	Input	Indicates the external queue is empty and cannot be read from.

Notes:

There is one set of these three interfaces per input queue instantiation.

TIE code: queue <name> <oqwifth> in

Table 10–31. TIE Lookup Signals (per Designer-Defined TIE Lookup)

Name (Configured Width)	I/O	Description
TIE_<name>_Out_Req	Output	Indicates that the processor is performing a lookup and that the TIE_<name>_Out interface has valid data.
TIE_<name>_Out [owidth-1:0]	Output	Output data bus of the TIE lookup, which can be used as an array index or simple address.
TIE_<name>_In [iwidth-1:0]	Input	Input data bus of the lookup, which is generally the data read from an external memory or array.
TIE_<name>_Rdy	Input	Optional TIE lookup arbitration signal to indicate the lookup request was not accepted by the external lookup memory or device.

Notes:

There is one set of these interfaces per lookup instantiation.

TIE code: lookup <name> {<owidth>, <ostage>} {<iwidth>, <istage>} [rdy]

Table 10–32. TIE Output FIFO32 Queue Signals (QIF32 Option)

Name (Configured Width)	I/O	Description
TIE_OPQ[31:0]	Output	Output data bus connected to the input of the external queue. The data is registered in the processor before it is sent to the external queue.
TIE_OPQ_PushReq	Output	Indicates that the processor is ready to write data to the output queue. This request is made independent of the status of the full signal and the attached queue is expected to ignore this signal if the queue is full.
TIE_OPQ_Full	Input	Indicates the external queue is full and cannot be written to.

Table 10–33. TIE Output Port (per Designer-Defined Export State)

Name (Configured Width)	I/O	Description
TIE_<name> [statewidth-1:0]	Output	Exported state data bus (or wire if 1-bit width). The architectural copy of the state is exported on these wires.
Notes: There is one interface per state that is exported TIE code: state <name> <statewidth> export		

Table 10–34. TIE Output Queue Signals (per Designer-Defined Output Queue)

Name (Configured Width)	I/O	Description
TIE_<name> [oqwidth-1:0]	Output	Output data bus (or wire if 1-bit width) connected to the input of the external queue. The data is registered in the processor before it is sent to the external queue.
TIE_<name>_PushReq	Output	Indicates that the processor is ready to write data to the output queue. This request is made independent of the status of the full signal and the attached queue is expected to ignore this signal if the queue is full.
TIE_<name>_Full	Input	Indicates the external queue is full and cannot be written to.
Notes: There is one set of these three interfaces per output queue instantiation. TIE code: queue <name> <oqwifth> out		

10.6 Processor Bus Interfaces

This section lists the PIF Master, PIF Slave, AHB, and AXI signal interfaces.

Table 10–35. PIF Master Signals (PIF and not AHB-Lite or AXI)

Name	I/O	Description	Notes
POReqValid*m*	Output	Indicates that there is a valid bus-transaction output request. All other signals prefixed with POReq are qualified by POReqValid	
POReqCntl*m*[7:0]	Output	Encodes bus-transaction request type and block size for block requests	
POReqAdrs*m*[31:0]	Output	Transaction request address. Address is aligned to the transfer size	
POReqData*m* [pifwidth-1:0]	Output	Data lines used by requests that require data during the request phase. Requests include single data write, block write, and read-conditional-write requests	
POReqDataBE*m* [pifwidth/8-1:0]	Output	Byte enables indicating valid lanes during requests that use POReqData, or byte lanes expected to be valid during responses that use PIRespData	
POReqId*m*[5:0]	Output	Request ID. Responses are expected to be tagged with the same ID as the request on PIRespId. IDs support multiple outstanding load and store requests. No restrictions or orderings are placed on the usage or uniqueness of IDs	
POReqAttribute*m* [11:0]	Output	PIF Request Attribute field if configured. Bottom 4 bits are user defined. Top 8 bits are memory attributes. The processor can set the bottom 4 bits with used defined load and store instructions	Requires PIF Request Attribute Option
POReqPriority*m* [1:0]		Request priority allows PIF slaves and interconnection networks to make priority judgments when arbitrating bus-transaction requests. 0x0 is low-priority, and 0x3 is high-priority	

Table 10–35. PIF Master Signals (PIF and not AHB-Lite or AXI) (continued)

Name	I/O	Description	Notes
PIReqRdy*m*	Input	Indicates that the PIF slave is ready to accept requests. A request transfer completes if PIReqRdy and POReqValid are both asserted during the same cycle	
PIRespValid*m*	Input	Indicates that there is a valid response. All other signals prefixed with PIResp are qualified by PIRespValid	
PIRespCntl*m*[7:0]	Input	Encodes response type and any error status for transaction requests	
PIRespData*m* [pifwidth-1:0]	Input	Response data. The data bus is equal in width to the request data bus	
PIRespId*m*[5:0]	Input	Response ID. Matches the ID of the corresponding request	
PIRespPriority*m*[1:0]	Input	Response priority. Matches the priority of the corresponding request	
PORespRdy*m*	Output	Indicates that the PIF master is ready to accept responses. A response transfer completes when PIRespValid and PORespRdy are asserted during the same cycle. PORespRdy can be toggled.	

Notes:

Depending on the driver of this PIF master interface, the suffix *m* can be blank or "_iDMA":

1. *m* is blank for the PIF master interface that is driven by the Xtensa core pipeline.

Table 10–36. PIF Slave Signals (Inbound-PIF Option, and not AHB-Lite or AXI)

Name (Configured Width)	I/O	Description	Notes
PIReqValid	Input	Indicates that there is a valid bus-transaction input request. All other signals prefixed with PIReq are qualified by PIReqValid.	
PIReqCntl[7:0]	Input	Encodes the bus-transaction request type and block size for block requests.	
PIReqAdrs[31:0]	Input	Transaction-request address. Address is aligned to the transfer size.	
PIReqData [pifwidth-1:0][1]	Input	Data lines used by requests that require data during the request phase. These requests include single data write, block write, and read-conditional-write requests. The data bus is configurable to 32, 64, or 128-bits.	
PIReqDataBE [pifwidth/8-1:0][1]	Input	Indicates valid bytes lanes during requests that use PIReqData, or byte lanes expected to be valid during responses that use PORespData.	
PIReqId[5:0]	Input	Request ID. Responses are expected to be tagged with the same ID as the request on PORespId. IDs support multiple outstanding requests. No restrictions or orderings are placed on the usage or uniqueness of IDs.	
PIReqPriority [1:0]	Input	Request priority allows PIF slaves and interconnection networks to make priority judgments when arbitrating bus-transaction requests. 0x0 is low-priority, and 0x3 is high-priority.	
PIReqAttribute [11:0]	Input	Request Attribute field. Bottom 4 bits are user defined. The processor does not use the request attribute information on its PIF slave port.	Requires PIF Request Attributes

Table 10–36. PIF Slave Signals (Inbound-PIF Option, and not AHB-Lite or AXI)

Name (Configured Width)	I/O	Description	Notes
POReqRdy	Output	Indicates that the slave is ready to accept requests. A request transfer completes if POReqRdy and PIReqValid are both asserted during the same cycle.	
PORespValid	Output	Indicates that there is a valid response. All other signals prefixed with POResp are qualified by PORespValid.	
PORespCntl[7:0]	Output	Encodes the response type and any error status on requests.	
PORespData [pifwidth-1:0][1]	Output	Response data. The data bus is configurable to 32, 64, or 128-bits and is equal in width to the request data bus.	
PORespId[5:0]	Output	Response ID. Matches the ID of the corresponding request.	
PORespPriority [1:0]	Output	Response priority. Matches the priority of the corresponding request.	
PIRespRdy	Input	Indicates that the master is ready to accept responses. A response transfer completes when PORespValid and PIRespRdy are asserted during the same cycle.	

Note:

1. pifwdith = configured PIF width, one of 32, 64, or 128 bits.

Table 10–37. AHB-Lite Master Interface (Requires AHB-Lite option)

Name (Configured Width)	I/O	Description	Notes
HTRANS[1:0]	Output	Transfer type.	
HWRITE	Output	High indicates write request, low indicates read request. Valid when HTRANS is non-idle.	
HLOCK	Output	Indicates requests should be locked on the AHB-Lite bus.	
HMASTLOCK	Output	The bridge generates a second lock signal 1-cycle after the HLOCK signal, to have the timing that an AHB arbiter would have.	
HBURST[2:0]	Output	Burst size.	
HSIZE[2:0]	Output	Size of each transfer.	
HPROT[3:0]	Output	Protection signals, driven from PIF Master POReqAttribute signal if it is configured, fixed to 4'b0011 otherwise.	
HADDR[31:0]	Output	System address.	
HWDATA [ahbwidth-1:0][1]	Output	Write data.	
HREADY	Input	When high, indicates bus has completed a transfer.	
HRESP[1:0]	Input	Transfer status. HRESP[1] must always be 0, i.e. the SPLIT and RETRY responses are not allowed.	
HRDATA [ahbwidth-1:0][1]	Input	Read data.	
HXTUSER[3:0]	Output	User-defined request attribute bits, same as the bottom 4 bits of PIF Master POReqAttribute signal. The processor can set them with user defined load and store instructions.	Requires PIF Request Attributes

Note:
1. ahbwidth = same as PIF width

Table 10–38. AHB-Lite Slave Interface (Requires Inbound PIF)

Name (Configured Width)	I/O	Description	Notes
HSEL_S	Input	Decoder select signal.	
HREADY_IN_S	Input	HREADY from bus, used to gate other control signals.	
HTRANS_S[1:0]	Input	Transfer type.	
HWRITE_S	Input	High indicates write request, low indicates read request. Valid when HTRANS_S is non-idle.	
HMASTLOCK_S	Input	Indicates requests should be lock the AHB-Lite bus. The bridge ignores the lock signal.	
HBURST_S[2:0]	Input	Burst size.	
HSIZE_S[2:0]	Input	Size of each transfer.	
HPROT_S[3:0]	Input	Protection signals, the bridge ignores the input protection.	
HADDR_S[31:0]	Input	System address.	
HWDATA_S [ahbwidth-1:0][1]	Input	Write data.	
HREADY_OUT_S	Output	When high, indicates bus has completed a transfer.	
HRESP_S[1:0]	Output	Transfer status.	
HRDATA_S[ahbwidth-1:0][1]	Output	Read data.	

Note:
1. ahbwidth = same as PIF width

Table 10–39. AXI Master Interface Signals

Name	I/O	Description
ARADDR[31:0]	Output	Read address channel address
ARBURST[1:0]	Output	Read address channel burst type
ARCACHE[3:0]	Output	Read address channel cache properties, driven from PIF master POReqAttribute signal if PIF Request Attributes are configured, driven to 4'd0 otherwise
ARID[3:0]	Output	Read address channel ID
ARLEN[len-1:0][2]	Output	Read address channel burst length
ARLOCK[lock_len-1:0][3]	Output	Read address channel lock control
ARPROT[2:0]	Output	Read address channel protection, driven from PIF master POReqAttribute signal if PIF Request Attributes are configured, driven to 3'd0 otherwise
ARQOS[3:0]	Output	Read address channel QOS (Requires AXI4)
ARSIZE[2:0]	Output	Read address channel burst size
ARVALID	Output	Read address channel request valid
ARREADY	Input	Read address channel request ready
ARXTUSER[3:0]	Output	User-defined request attribute bits, present only when PIF Request Attributes are configured. Same as the bottom 4 bits of the PIF Master POReqAttribute signal. The processor can set them with user-defined load and store instructions
RREADY	Output	Read response channel response ready
RDATA[axiwidth-1:0][1]	Input	Read response channel response data
RID[3:0]	Input	Read response channel ID
RLAST	Input	Read response channel last response
RRESP[1:0]	Input	Read response channel response control
RVALID	Input	Read response channel response valid

Table 10–39. AXI Master Interface Signals (continued)

Name	I/O	Description
AWADDR[31:0]	Output	Write address channel address
AWBURST[1:0]	Output	Write address channel burst type
AWCACHE[3:0]	Output	Write address channel cache properties, driven from PIF master POReqAttribute signal if PIF Request Attributes are configured, driven to 4'd0 otherwise
AWID[3:0]	Output	Write address channel ID
AWLEN[len-1:0][2]	Output	Write address channel burst length
AWLOCK[lock_len-1:0][3]	Output	Write address channel lock control
AWQOS[3:0]	Output	Write address channel QOS (Requires AXI4)
AWPROT[2:0]	Output	Write address channel protection, driven from PIF master POReqAttribute signal if PIF Request Attributes are configured, driven to 3'd0 otherwise
AWSIZE[2:0]	Output	Write address channel burst size
AWVALID	Output	Write address channel address valid
AWREADY	Input	Write address channel request ready
AWXTUSER[3:0]	Output	User-defined request attribute bits, only present when PIF Request Attributes are configured, same as the bottom 4 bits of the PIF Master POReqAttribute signal. The processor can set them with user-defined load and store instructions
WDATA[axiwidth-1:0][1]	Output	Write data channel write data
WID[3:0]	Output	Write data channel ID (Requires AXI3)
WLAST	Output	Write data channel last data transfer
WSTRB [axiwidth/8-1:0][1]	Output	Write data channel strobe
WVALID	Output	Write data channel data valid
WREADY	Input	Write data channel request ready
BREADY	Output	Write response channel response ready

Table 10–39. AXI Master Interface Signals (continued)

Name	I/O	Description
BID[3:0]	Input	Write response channel ID
BRESP[1:0]	Input	Write response channel response control
BVALID	Input	Write response channel response valid
ARSNOOP[3:0]	Output	Read address channel coherence signal, requires ACE-Lite
ARBAR[1:0]	Output	Read address channel barrier signal, requires ACE-Lite
ARDOMAIN[1:0]	Output	Read address channel domain signal, requires ACE-Lite
AWSNOOP[2:0]	Output	Write address channel coherence signal, requires ACE-Lite
AWBAR[1:0]	Output	Write address channel barrier signal, requires ACE-Lite
AWDOMAIN[1:0]	Output	Write address channel domain signal, requires ACE-Lite
AXISECMST	Input	Security interface signal for AXI Master, requires AXI security interface

Notes:

1.axiwidth = same as PIF width

2. len = 4 for AXI3, 8 for AXI4

3.lock_ len = 2 for AXI3, 1 for AXI4

Table 10–40. AXI4 Master Interface ECC/Parity Signals

Name	I/O	Description
ARADDRPTY[3:0]	Output	Parity bits for each byte in ARADDR signal: Bit 0 is parity bit for ARADDR[7:0] Bit 1 is parity bit for ARADDR[15:8] Bit 2 is parity bit for ARADDR[23:16] Bit 3 is parity bit for ARADDR[31:24]
ARCNTLPTY[11:0]	Output	Parity bits for other control signals in the read address channel: Bit 0 is parity bit for ARID Bit 1 is parity bit for ARLEN Bit 2 is parity bit for ARSIZE Bit 3 is parity bit for ARBURST Bit 4 is parity bit for ARLOCK Bit 5 is parity bit for ARCACHE Bit 6 is parity bit for ARPROT Bit 7 is parity bit for ARQOS (if AXI4 configured, tied to 0 otherwise) Bit 8 is parity bit for ARXTUSER (if PIF reqAttr configured, tied to 0 otherwise) Bit 9 is parity bit for ARSNOOP (if ACE-Lite is configured, tied to 0 otherwise) Bit 10 is parity bit for ARBAR (if ACE-Lite is configured, tied to 0 otherwise) Bit 11 is parity bit for ARDOMAIN (if ACE-Lite is configured, tied to 0 otherwise)
ARVPTY	Output	Parity bit for ARVALID signal
ARRPTY	Input	Parity bit for ARREADY signal
RRPTY	Output	Parity bit for RREADY signal
RCNTLPTY[2:0]	Input	Parity bits for control signals in the read response channel: Bit 0 is parity bit for RID Bit 1 is parity bit for RRESP Bit 2 is parity bit for RLAST
RDATAECC [ecc_width-1:0][4]	Input	ECC code for RDATA signal
RVPTY	Input	Parity bit for RVALID signal

Table 10–40. AXI4 Master Interface ECC/Parity Signals (continued)

Name	I/O	Description
AWADDRPTY[3:0]	Output	Parity bits for each byte in AWADDR signal: Bit 0 is parity bit for AWADDR[7:0] Bit 1 is parity bit for AWADDR[15:8] Bit 2 is parity bit for AWADDR[23:16] Bit 3 is parity bit for AWADDR[31:24]
AWCNTLPTY[11:0]	Output	Parity bits for other control signals in the write address channel: Bit 0 is parity bit for AWID Bit 1 is parity bit for AWLEN Bit 2 is parity bit for AWSIZE Bit 3 is parity bit for AWBURST Bit 4 is parity bit for AWLOCK Bit 5 is parity bit for AWCACHE Bit 6 is parity bit for AWPROT Bit 7 is parity bit for AWQOS (if AXI4 configured, tied to 0 otherwise) Bit 8 is parity bit for AWXTUSER (if PIF reqAttr configured, tied to 0 otherwise) Bit 9 is parity bit for AWSNOOP (if ACE-Lite is configured, tied to 0 otherwise) Bit 10 is parity bit for AWBAR (if ACE-Lite is configured, tied to 0 otherwise) Bit 11 is parity bit for AWDOMAIN (if ACE-Lite is configured, tied to 0 otherwise)
AWVPTY	Output	Parity bit for AWVALID signal
AWRPTY	Input	Parity bit for AWREADY signal
WCNTLPTY[1:0]	Output	Parity bits for control signals in the write data channel: Bit 0 is parity bit for WLAST Bit 1 is parity bit for WID (if AXI3 is configured, tied to 0 otherwise)
WDATAECC [ecc_width-1:0][4]	Input	ECC code for WDATA signal
WSTRBPTY [strb_parity_width-1:0][5]	Output	Parity bits for WSTRB signal
WVPTY	Output	Parity bit for WVALID signal
WRPTY	Input	Parity bit for WREADY signal

Table 10–40. AXI4 Master Interface ECC/Parity Signals (continued)

Name	I/O	Description
BRPTY	Output	Parity bit for BREADY signal
BCNTLPTY[1:0]	Input	Parity bits for control signals in the write response channel: Bit 0 is parity bit for BID Bit 1 is parity bit for BRESP
BVPTY	Input	Parity bit for BVALID signal
AXIPARITYSEL	Input	Signal to select parity type (even/odd) for AXI Master and slave, and iDMA Master port.

Notes:

1.All the signals listed in this table require AXI Parity/ECC protection to be configured

2. Even parity if AXIPARITYSEL=1'b0 or odd parity if AXIPARITYSEL=1'b1 for the following signals: ARADDRPTY[3:0], ARCNTLPTY[11;0], RCNTLPTY[2:0], AWADDRPTY[3:0], AWCNTLPTY[11:0], WCNTLPTY[1:0], and BCNTLPTY[1:0]

3. Always odd parity regardless of the value of the AXIPARITYSEL signal for the following signals: ARVPTY, ARRPTY, RRPTY, RVPTY, AWVPTY, AWRPTY, WVPTY, WRPTY,BRPTY, and BVPTY.

4. ecc_width = 7 * (axiwidth / 32)

5.srtb_parity_width = 2 if axiwidth is 128, 1 otherwise

Table 10–41. AXI Slave Interface

Name (Configured Width)	I/O	Description
ARADDR_S[31:0]	Input	Read address channel address
ARBURST_S[1:0]	Input	Read address channel burst type
ARCACHE_S[3:0]	Input	Read address channel cache properties. This field is unused
ARQOS_S[3:0]	Input	Read address channel QOS. (Requires AXI4)
ARID_S[15:0]	Input	Read address channel ID
ARLEN_S[len:0][2]	Input	Read address channel length
ARLOCK_S[lock_len-1:0][3]	Input	Read address channel lock control
ARPROT_S[2:0]	Input	Read address channel protection.
ARSIZE_S[2:0]	Input	Read address channel size
ARVALID_S	Input	Read address channel request valid
ARREADY_S	Output	Read address channel request ready
RREADY_S	Input	Read response channel response ready
RDATA_S [axiwidth-1:0][1]	Output	Read response channel response data

Table 10–41. AXI Slave Interface (continued)

Name (Configured Width)	I/O	Description
RID_S[15:0]	Output	Read response channel ID
RLAST_S	Output	Read response channel last response
RRESP_S[1:0]	Output	Read response channel response control
RVALID_S	Output	Read response channel response valid
AWADDR_S[31:0]	Input	Write address channel address
AWBURST_S[1:0]	Input	Write address channel burst type
AWCACHE_S[3:0]	Input	Write address channel cache properties. This field is unused.
AWQOS_S[3:0]	Input	Write address channel QOS. (Requires AXI4)
AWID_S[15:0]	Input	Write address channel ID
AWLEN_S[len:0][2]	Input	Write address channel length
AWLOCK_S[lock_len-1:0][3]	Input	Write address channel lock control
AWPROT_S[2:0]	Input	Write address channel protection.
AWSIZE_S[2:0]	Input	Write address channel size
AWVALID_S	Input	Write address channel address valid
AWREADY_S	Output	Write address channel request ready
WDATA_S [axiwidth-1:0][1]	Input	Write data channel write data
WID_S[15:0]	Input	Write data channel ID (Requires AXI3)
WLAST_S	Input	Write data channel last data transfer
WSTRB_S [axiwidth/8-1:0][1]	Input	Write data channel strobe
WVALID_S	Input	Write data channel data valid
WREADY_S	Output	Write data channel request ready
BREADY_S	Input	Write response channel response ready
BID_S[15:0]	Output	Write response channel ID
BRESP_S[1:0]	Output	Write response channel response control

Table 10–41. AXI Slave Interface (continued)

Name (Configured Width)	I/O	Description
BVALID_S	Output	Write response channel response valid
AXISECSLV	Input	Security interface signal for AXI Master, requires AXI security interface

Notes:
1. axiwidth = same as PIF width
2. len = 4 for AXI3, 8 for AXI4
3. lock_len = 2 for AXI3, 1for AXI4
Note that all AXI slave interfaces require Inbound PIF

Table 10–42. AXI4 Slave Interface ECC/Parity Signals

Name	I/O	Description
ARADDRPTY_S[3:0]	Input	Parity bits for each byte in ARADDR_S signal: Bit 0 is parity bit for ARADDR_S[7:0] Bit 1 is parity bit for ARADDR_S[15:8] Bit 2 is parity bit for ARADDR_S[23:16] Bit 3 is parity bit for ARADDR_S[31:24]
ARCNTLPTY_S[11:0]	Input	Parity bits for other control signals in the read address channel: Bit 0 is parity bit for ARID_S Bit 1 is parity bit for ARLEN_S Bit 2 is parity bit for ARSIZE_S Bit 3 is parity bit for ARBURST_S Bit 4 is parity bit for ARLOCK_S Bit 5 is parity bit for ARCACHE_S Bit 6 is parity bit for ARPROT_S Bit 7 is parity bit for ARQOS_S (if AXI4 configured, tied to 0 otherwise) Bit 8 is parity bit for ARXTUSER_S (if PIF reqAttr configured, tied to 0 otherwise) Bit 9 is parity bit for ARSNOOP_S (if ACE-Lite is configured, tied to 0 otherwise) Bit 10 is parity bit for ARBAR_S (if ACE-Lite is configured, tied to 0 otherwise) Bit 11 is parity bit for ARDOMAIN_S (if ACE-Lite is configured, tied to 0 otherwise)
ARVPTY_S	Input	Parity bit for ARVALID_S signal
ARRPTY_S	Output	Parity bit for ARREADY_S signal
RRPTY_S	Input	Parity bit for RREADY_S signal
RCNTLPTY_S[2:0]	Output	Parity bits for control signals in the read response channel: Bit 0 is parity bit for RID_S Bit 1 is parity bit for RRESP_S Bit 2 is parity bit for RLAST_S
RDATAECC_S [ecc_width-1:0][4]	Output	ECC code for RDATA_S signal
RVPTY_S	Output	Parity bit for RVALID_S signal

Table 10–42. AXI4 Slave Interface ECC/Parity Signals (continued)

Name	I/O	Description
AWADDRPTY_S[3:0]	Input	Parity bits for each byte in AWADDR_S signal: Bit 0 is parity bit for AWADDR_S[7:0] Bit 1 is parity bit for AWADDR_S[15:8] Bit 2 is parity bit for AWADDR_S[23:16] Bit 3 is parity bit for AWADDR_S[31:24]
AWCNTLPTY_S[11:0]	Input	Parity bits for other control signals in the write address channel: Bit 0 is parity bit for AWID_S Bit 1 is parity bit for AWLEN_S Bit 2 is parity bit for AWSIZE_S Bit 3 is parity bit for AWBURST_S Bit 4 is parity bit for AWLOCK_S Bit 5 is parity bit for AWCACHE_S Bit 6 is parity bit for AWPROT_S Bit 7 is parity bit for AWQOS_S (if AXI4 configured, tied to 0 otherwise) Bit 8 is parity bit for AWXTUSER_S (if PIF reqAttr configured, tied to 0 otherwise) Bit 9 is parity bit for AWSNOOP_S (if ACE-Lite is configured, tied to 0 otherwise) Bit 10 is parity bit for AWBAR_S (if ACE-Lite is configured, tied to 0 otherwise) Bit 11 is parity bit for AWDOMAIN_S (if ACE-Lite is configured, tied to 0 otherwise)
AWVPTY_S	Input	Parity bit for AWVALID_S signal
AWRPTY_S	Output	Parity bit for AWREADY_S signal
WCNTLPTY_S[1:0]	Input	Parity bits for control signals in the write data channel: Bit 0 is parity bit for WLAST_S Bit 1 is parity bit for WID_S (if AXI3 is configured, tied to 0 otherwise)
WDATAECC_S [ecc_width-1:0][4]	Output	ECC code for WDATA_S signal
WSTRBPTY_S [strb_parity_width-1:0][5]	Input	Parity bits for WSTRB_S signal
WVPTY_S	Input	Parity bit for WVALID_S signal
WRPTY_S	Output	Parity bit for WREADY_S signal

Table 10–42. AXI4 Slave Interface ECC/Parity Signals (continued)

Name	I/O	Description
BRPTY_S	Input	Parity bit for BREADY_S signal
BCNTLPTY_S[1:0]	Output	Parity bits for control signals in the write response channel: Bit 0 is parity bit for BID_S Bit 1 is parity bit for BRESP_S
BVPTY_S	Output	Parity bit for BVALID_S signal

Notes:

1.All the signals listed in this table require AXI Parity/ECC protection to be configured

2. Even parity if AXIPARITYSEL=1'b0 or odd parity if AXIPARITYSEL=1'b1 for the following signals: ARADDRPTY_S[3:0], ARCNTLPTY_S[11;0], RCNTLPTY_S[2:0], AWADDRPTY_S[3:0], AWCNTLPTY_S[11:0], WCNTLPTY_S[1:0], and BCNTLPTY_S[1:0]

3. Always odd parity regardless of the value of the AXIPARITYSEL signal for the following signals: ARVPTY_S, ARRPTY_S, RRPTY_S, RVPTY_S, AWVPTY_S, AWRPTY_S, WVPTY_S, WRPTY_S,BRPTY_S, and BVPTY_S.

4. ecc_width = 7 * (axiwidth / 32)

5.srtb_parity_width = 2 if axiwidth is 128, 1 otherwise

10.7 iDMA Bus Interfaces

This section lists the integrated DMA AXI and port interface signals.

Table 10–43. iDMA AXI4 Master Interface Signals

Name	I/O	Description
ARADDR_iDMA[31:0]	Output	Read address channel address
ARBURST_iDMA[1:0]	Output	Read address channel burst type
ARCACHE_iDMA[3:0]	Output	Read address channel cache properties, driven from PIF master POReqAttribute signal if PIF Request Attributes are configured, driven to 4'd0 otherwise
ARID_iDMA[3:0]	Output	Read address channel ID
ARLEN_iDMA[len-1:0][2]	Output	Read address channel burst length
ARLOCK_iDMA [lock_len-1:0][3]	Output	Read address channel lock control
ARPROT_iDMA[2:0]	Output	Read address channel protection, driven from PIF master POReqAttribute signal if PIF Request Attributes are configured, driven to 3'd0 otherwise
ARQOS_iDMA[3:0]	Output	Read address channel QOS (Requires AXI4)
ARSIZE_iDMA[2:0]	Output	Read address channel burst size
ARVALID_iDMA	Output	Read address channel request valid
ARREADY_iDMA	Input	Read address channel request ready
ARXTUSER_iDMA[3:0]	Output	User-defined request attribute bits, present only when PIF Request Attributes are configured. Same as the bottom 4 bits of the PIF Master POReqAttribute signal. The processor can set them with user-defined load and store instructions
RREADY_iDMA	Output	Read response channel response ready
RDATA_iDMA [axiwidth-1:0][1]	Input	Read response channel response data
RID_iDMA[3:0]	Input	Read response channel ID
RLAST_iDMA	Input	Read response channel last response

Table 10–43. iDMA AXI4 Master Interface Signals (continued)

Name	I/O	Description
RRESP_iDMA[1:0]	Input	Read response channel response control
RVALID_iDMA	Input	Read response channel response valid
AWADDR_iDMA[31:0]	Output	Write address channel address
AWBURST_iDMA[1:0]	Output	Write address channel burst type
AWCACHE_iDMA[3:0]	Output	Write address channel cache properties, driven from PIF master POReqAttribute signal if PIF Request Attributes are configured, driven to 4'd0 otherwise
AWID_iDMA[3:0]	Output	Write address channel ID
AWLEN_iDMA[len-1:0][2]	Output	Write address channel burst length
AWLOCK_iDMA [lock_len-1:0][3]	Output	Write address channel lock control
AWQOS_iDMA[3:0]	Output	Write address channel QOS (Requires AXI4)
AWPROT_iDMA[2:0]	Output	Write address channel protection, driven from PIF master POReqAttribute signal if PIF Request Attributes are configured, driven to 3'd0 otherwise
AWSIZE_iDMA[2:0]	Output	Write address channel burst size
AWVALID_iDMA	Output	Write address channel address valid
AWREADY_iDMA	Input	Write address channel request ready
AWXTUSER_iDMA[3:0]	Output	User-defined request attribute bits, only present when PIF Request Attributes are configured, same as the bottom 4 bits of the PIF Master POReqAttribute signal. The processor can set them with user-defined load and store instructions
WDATA_iDMA [axiwidth-1:0][1]	Output	Write data channel write data
WID_iDMA[3:0]	Output	Write data channel ID
WLAST_iDMA	Output	Write data channel last data transfer
WSTRB_iDMA [axiwidth/8-1:0][1]	Output	Write data channel strobe

Table 10–43. iDMA AXI4 Master Interface Signals (continued)

Name	I/O	Description
WVALID_iDMA	Output	Write data channel data valid
WREADY_iDMA	Input	Write data channel request ready
BREADY_iDMA	Output	Write response channel response ready
BID_iDMA[3:0]	Input	Write response channel ID
BRESP_iDMA[1:0]	Input	Write response channel response control
BVALID_iDMA	Input	Write response channel response valid
ARSNOOP_iDMA[3:0]	Output	Read address channel coherence signal, requires ACE-Lite
ARBAR_iDMA[1:0]	Output	Read address channel barrier signal, requires ACE-Lite
ARDOMAIN_iDMA[1:0]	Output	Read address channel domain signal, requires ACE-Lite
AWSNOOP_iDMA[2:0]	Output	Write address channel coherence signal, requires ACE-Lite
AWBAR_iDMA[1:0]	Output	Write address channel barrier signal, requires ACE-Lite
AWDOMAIN_iDMA[1:0]	Output	Write address channel domain signal, requires ACE-Lite

Notes:
1. axiwidth = same as PIF width
2. len = 4 for AXI3, 8 for AXI4
3. lock_len = 2 for AXI3, 1for AXI4

Table 10–44. iDMA AXI4 Master Interface ECC/Parity Signals

Name	I/O	Description
ARADDRPTY_iDMA[3:0]	Output	Parity bits for each byte in ARADDR signal: Bit 0 is parity bit for ARADDR_iDMA[7:0] Bit 1 is parity bit for ARADDR_iDMA[15:8] Bit 2 is parity bit for ARADDR_iDMA[23:16] Bit 3 is parity bit for ARADDR_iDMA[31:24]
ARCNTLPTY_iDMA[11:0]	Output	Parity bits for other control signals in the read address channel: Bit 0 is parity bit for ARID_iDMA Bit 1 is parity bit for ARLEN_iDMA Bit 2 is parity bit for ARSIZE_iDMA Bit 3 is parity bit for ARBURST_iDMA Bit 4 is parity bit for ARLOCK_iDMA Bit 5 is parity bit for ARCACHE_iDMA Bit 6 is parity bit for ARPROT_iDMA Bit 7 is parity bit for ARQOS_iDMA (if AXI4 configured, tied to 0 otherwise) Bit 8 is parity bit for ARXTUSER_iDMA (if PIF reqAttr configured, tied to 0 otherwise) Bit 9 is parity bit for ARSNOOP_iDMA (if ACE-Lite is configured, tied to 0 otherwise) Bit 10 is parity bit for ARBAR_iDMA (if ACE-Lite is configured, tied to 0 otherwise) Bit 11 is parity bit for ARDOMAIN_iDMA (if ACE-Lite is configured, tied to 0 otherwise)
ARVPTY_iDMA	Output	Parity bit for ARVALID signal
ARRPTY_iDMA	Input	Parity bit for ARREADY signal
RRPTY_iDMA	Output	Parity bit for RREADY signal
RCNTLPTY_iDMA[2:0]	Input	Parity bits for control signals in the read response channel: Bit 0 is parity bit for RID_iDMA Bit 1 is parity bit for RRESP_iDMA Bit 2 is parity bit for RLAST_iDMA
RDATAECC_iDMA [ecc_width-1:0][4]	Input	ECC code for RDATA signal
RVPTY_iDMA	Input	Parity bit for RVALID signal

Table 10–44. iDMA AXI4 Master Interface ECC/Parity Signals (continued)

Name	I/O	Description
AWADDRPTY_iDMA[3:0]	Output	Parity bits for each byte in AWADDR signal: Bit 0 is parity bit for AWADDR_iDMA[7:0] Bit 1 is parity bit for AWADDR_iDMA[15:8] Bit 2 is parity bit for AWADDR_iDMA[23:16] Bit 3 is parity bit for AWADDR_iDMA[31:24]
AWCNTLPTY_iDMA[11:0]	Output	Parity bits for other control signals in the write address channel: Bit 0 is parity bit for AWID_iDMA Bit 1 is parity bit for AWLEN_iDMA Bit 2 is parity bit for AWSIZE_iDMA Bit 3 is parity bit for AWBURST_iDMA Bit 4 is parity bit for AWLOCK_iDMA Bit 5 is parity bit for AWCACHE_iDMA Bit 6 is parity bit for AWPROT_iDMA Bit 7 is parity bit for AWQOS_iDMA (if AXI4 configured, tied to 0 otherwise) Bit 8 is parity bit for AWXTUSER_iDMA (if PIF reqAttr configured, tied to 0 otherwise) Bit 9 is parity bit for AWSNOOP_iDMA (if ACE-Lite is configured, tied to 0 otherwise) Bit 10 is parity bit for AWBAR_iDMA (if ACE-Lite is configured, tied to 0 otherwise) Bit 11 is parity bit for AWDOMAIN_iDMA (if ACE-Lite is configured, tied to 0 otherwise)
AWVPTY_iDMA	Output	Parity bit for AWVALID signal
AWRPTY_iDMA	Input	Parity bit for AWREADY signal
WCNTLPTY_iDMA[1:0]	Output	Parity bits for control signals in the write data channel: Bit 0 is parity bit for WLAST_iDMA Bit 1 is parity bit for WID_iDMA (if AXI3 is configured, tied to 0 otherwise)
WDATAECC_iDMA [ecc_width-1:0][4]	Input	ECC code for WDATA signal
WSTRBPTY_iDMA [strb_parity_width-1:0][5]	Output	Parity bits for WSTRB signal
WVPTY_iDMA	Output	Parity bit for WVALID signal
WRPTY_iDMA	Input	Parity bit for WREADY signal

Table 10–44. iDMA AXI4 Master Interface ECC/Parity Signals (continued)

Name	I/O	Description
BRPTY_iDMA	Output	Parity bit for BREADY signal
BCNTLPTY_iDMA[1:0]	Input	Parity bits for control signals in the write response channel: Bit 0 is parity bit for BID_iDMA Bit 1 is parity bit for BRESP_iDMA
BVPTY_iDMA	Input	Parity bit for BVALID signal

Notes:

1.All the signals listed in this table require AXI Parity/ECC protection to be configured

2. Even parity if AXIPARITYSEL=1'b0 or odd parity if AXIPARITYSEL=1'b1 for the following signals: ARADDRPTY_iDMA[3:0], ARCNTLPTY_iDMA[11;0], RCNTLPTY_iDMA[2:0], AWADDRPTY_iDMA[3:0], AWCNTLPTY_iDMA[11:0], WCNTLPTY_iDMA[1:0], and BCNTLPTY_iDMA[1:0]

3. Always odd parity regardless of the value of the AXIPARITYSEL signal for the following signals: ARVPTY_iDMA, ARRPTY_iDMA, RRPTY_iDMA, RVPTY_iDMA, AWVPTY_iDMA, AWRPTY_iDMA, WVPTY_iDMA, WRPTY_iDMA, BRPTY_iDMA, and BVPTY_iDMA.

4. ecc_width = 7 * (axiwidth / 32)

5.srtb_parity_width = 2 if axiwidth is 128, 1 otherwise

Table 10–45. iDMA Port Signals

Name (Configured Width)	I/O	Description
TrigIn_iDMA	Input	The trigger input that increases the iDMA trigger count when it is asserted.
TrigOut_iDMA	Output	Indicates the completion of an iDMA descriptor that is configured to send trigger out upon finish.

10.8 Fault Status Interface

Table 10–46 lists the fault reporter status interface signals.

Table 10–46. Fault Reporter

Name (Configured Width)	I/O	Description
DoubleExceptionError	Output	Single cycle assertion for every time double exception faults occur.
PFatalError	Output	Sticky fatal error notification signal that is asserted when a fatal error condition occurs.

PFaultInfo[31:0]	Output	Fault information signal. This signal mirrors the internal Fault Information Register and provides the source and severity of the fault.
PFaultInfoValid	Output	Strobe signal that is asserted for one cycle every time PFaultInfo signal changes its value.

Table 10–47 lists coprocessor exceptions that are signaled outside of the processor.

Table 10–47. Coprocessor Exceptions

Name (Configured Width)	I/O	Description
PArithmeticException	Output	DSP has encountered an arithmetic exception. Note: For Vision P5, Vision P6, Fusion G3, ConnX BBE16EP, ConnX BBE32EP, or ConnX BBE64E DSPs only.

10.9 Debug Interfaces

This section lists the APB, Test, On-Chip Debug, JTAG, Traceport, and TRAX interface signals. For more details on these interfaces, refer to the *Xtensa Debug Guide*.

Table 10–48. APB Interface Signals

Name	I/O	Description	Notes
PBCLK	Input	Clock used for APB transfers	Requires: APB
PBCLKEN	Input	Qualifies the rising edge of PBCLK	Requires: APB
PRESETn	Input	Resets the APB interface. Active low	Requires: APB
PADDR[31:0]	Input	Address of the APB access	Requires: APB
PSEL	Input	Indicates that Xtensa is selected and APB access is required	Requires: APB
PENABLE	Input	Indicates the second and subsequent cycles of the APB access	Requires: APB
PWRITE	Input	Indicates a write access when HIGH and read access when LOW	Requires: APB
PWDATA[31:0]	Input	Data of the write access	Requires: APB
PREADY	Output	Used by Xtensa to extend the number of cycles of the APB access	Requires: APB
PRDATA[31:0]	Output	Data of the read access	Requires: APB
PSLVERR	Output	Indicates an error on the access	Requires: APB

Table 10–49. Test and On-Chip Debug Interface Signals

Name	I/O	Description	Notes
DReset	Input	Debug Reset; used to reset debug functionality separately from the Xtensa core	Requires: OCD or PSO
OCDHaltOnReset	Input	Enters OCDHaltMode if this signal is sampled asserted on reset	Requires: OCD
XOCDMode	Output	Indicates that the processor is in OCD halt mode	Requires: OCD
DebugMode	Output	Same as XOCDMode but not maskable by software	Requires: OCD
BreakOut	Output	Indication from the processor that it has entered OCD halt mode	Requires: OCD
BreakIn	Input	External debug interrupt	Requires: OCD

Table 10–49. Test and On-Chip Debug Interface Signals (continued)

Name	I/O	Description	Notes
BreakOutAck	Input	Acknowledges that BreakOut has been received. Clears BreakOut	Requires: OCD
BreakInAck	Output	Acknowledges that BreakIn has been received	Requires: OCD
DBGEN	Input	Non-secure, invasive debug enable. This function is as described by the ARM CoreSight specifications.	Requires: APB
NIDEN	Input	Non-secure, non-invasive debug enable. This function is as described by the ARM CoreSight specifications.	Requires: APB
SPIDEN	Input	Secure, invasive debug enable. This function is as described by the ARM CoreSight specifications.	Requires: APB
SPNIDEN	Input	Secure, non-invasive debug enable. This function is as described by the ARM CoreSight specifications.	Requires: APB
TMode	Input	Special test mode signal for scan testing. It controls async-reset (including JTRST)	Requires: Scan
TModeClkGateOverride	Input	Special test mode signal to override clock gating during scan testing	Requires: Scan and clock gating (global or functions)

Table 10–50. JTAG Interface Signals

Name	I/O	Description	Notes
JTRST	Input	Standard 119.1 JTAG Test Reset	Requires: OCD or PSO
JTCK	Input	Standard 1149.1 JTAG TAP clock	Requires: OCD or PSO
JTDI	Input	Standard 1149.1 JTAG TAP port serial data input	Requires: OCD or PSO
JTMS	Input	Standard 1149.1 JTAG Test Mode Select	Requires: OCD or PSO
JTDO	Output	Standard 1149.1 JTAG Test Data Output	Requires: OCD or PSO
JTDOEn	Output	JTAG TDO Output Enable	Requires: OCD or PSO

Table 10–51. Traceport Signals

Name (Configured Width)	I/O	Description	Notes
PDebugEnable	Input	Traceport enable	Requires Trace
PDebugInst[31:0]	Output	Instruction information	Requires Trace
PDebugStatus[7:0]	Output	Traceport status information	Requires Trace
PDebugData[31:0]	Output	Trace debug data	Requires Trace
PDebugPC[31:0]	Output	Trace program counter	Requires Trace
PDebugLS{0,1,2}Stat[31:0]	Output	LSU {0,1,2} status	Requires Data Trace or PerfMon
PDebugLS{0,1,2}Addr[31:0]	Output	LSU {0,1,2} address	Requires Data Trace
PDebugLS{0,1,2}Data[31:0]	Output	LSU {0,1,2} data	Requires Data Trace
PDebugOutPif[15:0]	Output	Outbound PIF transaction information	Requires Trace,Writeback DCache OR Prefetch, PerfMon
PDebugInbPif[7:0]	Output	Inbound PIF transaction information	Requires Trace, Inbound PIF, PerfMon
PDebugPrefetchLookup[7:0]	Output	Prefetch lookup information	Requires Trace, Prefetch, PerfMon
PDebugPrefetchL1Fill[3:0]	Output	Prefetch L1 fill information	Requires Trace, Prefetch, PerfMon
PDebugiDMA[7:0]	Output	iDMA transaction information	Requires Trace, iDMA, PerfMon

Table 10–52. TRAX Interface Signals

Name	I/O	Description	Notes
CrossTriggerIn	Input	Trigger in from other core or logic to stop tracing	Requires: TRAX
CrossTriggerOut	Output	Trigger out to other core or logic indicating that tracing has been stopped	Requires: TRAX
CrossTriggerInAck	Output	Acknowledges that CrossTriggerIn has been received	Requires: TRAX
CrossTriggerOutAck	Input	Acknowledges that CrossTriggerOut has been received. Clears CrossTriggerOut	Requires: TRAX
TraceMemReady	Input	TraceRAM is ready to accept a read or write for this cycle	Requires: TRAX and TraceRAM sharing

Table 10–52. TRAX Interface Signals (continued)

Name	I/O	Description	Notes
TraceMemAddr [log_memory_size-3:0]	Output	TraceRAM address	Requires: OCD, TRAX
TraceMemData[31:0]	Input	Input data from TraceRAM	Requires: OCD, TRAX
TraceMemEn	Output	TraceRAM enable	Requires: OCD, TRAX
TraceMemWr	Output	TraceRAM write enable	Requires: OCD, TRAX
TraceMemWrData [31:0]	Output	Trace write data	Requires: OCD, TRAX
ATCLK	Input	Meant to be ATB clock, but not implemented. Clock within TRAX used. ATCLK is an unconnected input pin	Requires: TRAX and APB
ATCLKEN	Input	Clock enable for ATB-related logic	Requires: TRAX and APB
ATRESETn	Input	Meant to be ATB reset, but not implemented. DReset to TRAX used to reset ATB-related logic. ATRESETn is only used to quiet (i.e. data-gate) ATBYTES and ATDATA	Requires: TRAX and APB
ATVALID	Output	Valid transfer request	Requires: TRAX and APB
ATREADY	Input	Trace sink is ready to accept data	Requires: TRAX and APB
ATID[6:0]	Output	An ID that uniquely identifies the source of the trace. From TRAX Control register bits [30:24]	Requires: TRAX and APB
ATBYTES[1:0]	Output	The number of bytes on ATDATA to be captured minus 1. TRAX always writes words, so value is only “11” or “00”	Requires: TRAX and APB
ATDATA[31:0]	Output	Trace data. Width is fixed at 32 bits – same as to TraceRAM	Requires: TRAX and APB
AFVALID	Input	Flush valid	Requires: TRAX and APB
AFREADY	Output	Flush acknowledge	Requires: TRAX and APB

10.10 Power Shut-Off Interface

Table 10–53 lists the power control module interface signals.

Table 10–53. Power Control Module Interface Signals

Name	I/O	Description	Notes
PsoCachesLostPower	Input	Notifies Xtensa that memory power domain was power cycled while Xtensa was powered off – caches are cold	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoWakeupReset	Input	When high, notifies Xtensa that it is coming out of shutdown, not starting from power-on reset	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoCoreStillNeeded	Input	When high, notifies the Xtensa power shutdown software sequence that another agent still needs the core, thus preventing shutdown	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoBResetProc	Input	Global BReset signal for the core power domain from PCM; wired to BReset except during core domain power up/down, when PCM module controls it	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoBResetDebug	Input	Global BReset signal for the debug power domain from PCM; wired to BReset except during debug domain power up/down, when PCM module controls it	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
PsoDResetDebug	Input	Debug reset signal from PCM, normally wired to PsoDResetPCM except during debug domain power up/down, when PCM module controls it	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
PsoDResetPCM	Output	Debug reset signal from Access Port to PCM	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
PsoBResetPCM	Output	Global BReset signal from Access Port to PCM	PSODomains = {single, core_debug_mem}, PSOCoreRetention = {none, partial, full}

Table 10–53. Power Control Module Interface Signals (continued)

Name	I/O	Description	Notes
PsoReqShutOffDebug	Input	Signal from PCM to request start of debug domain power down	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
PsoAckShutOffDebug	Output	Signal to PCM to allow debug domain power down to begin	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
PsoEriMemWakeup	Output	From Access Port to PCM so Xtensa can request wakeup of memory power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoJtagProcWakeup	Output	From Access Port to PCM so JTAG can request wakeup of core power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, JTAG access configured
PsoJtagMemWakeup	Output	From Access Port to PCM so that JTAG can request wakeup of memory power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, JTAG access configured
PsoApbProcWakeup	Output	From Access Port to PCM so that APB can request wakeup of core power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, APB access configured
PsoApbMemWakeup	Output	From Access Port to PCM so that APB can request wakeup of memory power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, APB access configured
PsoEriDebugWakeup	Output	From Access Port to PCM so Xtensa can request wakeup of debug power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured

Table 10–53. Power Control Module Interface Signals (continued)

Name	I/O	Description	Notes
PsoJtagDebugWakeup	Output	From Access Port to PCM so that JTAG can request wakeup of debug power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, JTAG access configured
PsoApbDebugWakeup	Output	From Access Port to PCM so that APB can request wakeup of debug power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, APB access configured
PsoExternalProcWakeup	Input	External system signal into Access Port to request wakeup of core power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoExternalProcWakeup PCM	Output	Signal from Access Port to PCM to request wakeup of core power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoExternalMemWakeup	Input	External system signal into Access Port to request wakeup of memory power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoExternalMemWakeup PCM	Output	Signal from Access Port to PCM to request wakeup of memory power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoExternalDebugWakeup	Input	External system signal into Access Port to request wakeup of debug power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
PsoExternalDebugWakeup PCM	Output	Signal from Access Port to PCM to request wakeup of debug power domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
PsoRailFeedbackProc	Output	Feedback signal from power switches of core domain to PCM – created during physical implementation	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}

Table 10–53. Power Control Module Interface Signals (continued)

Name	I/O	Description	Notes
PsoIsolateProc	Input	Control signal for isolation cells for core power domain – created during physical implementation	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoBretainProc	Input	Control signal for single-pin retention flops – created during physical implementation	PSODomains = core_debug_mem, PSOCoreRetention = {partial, full}
PsoSaveProc	Input	Save signal for dual-pin retention flops – created during physical implementation	PSODomains = core_debug_mem, PSOCoreRetention = {partial, full}
PsoBrestoreProc	Input	Restore signal for dual-pin retention flops – created during physical implementation	PSODomains = core_debug_mem, PSOCoreRetention = {partial, full}
PsoRemovePowerProc	Input	Control signal for power switch cells for core power domain – created during physical implementation	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoRailFeedbackDebug	Output	Feedback signal from power switches of debug domain to PCM – created during physical implementation	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
PsoIsolateDebug	Input	Control signal for isolation cells for debug power domain – created during physical implementation	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
PsoRemovePowerDebug	Input	Control signal for power switch cells for debug power domain – created during physical implementation	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
PsoShutProcOffOnPWait	Output	Indicates to PCM that Xtensa has begun software shutdown routine; when both PWaitMode and this signal are high, PCM begins powering down core domain	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}

Table 10–53. Power Control Module Interface Signals (continued)

Name	I/O	Description	Notes
PsoDomainOffProc	Input	Tells the outside world and Access Port that the core power domain is off	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoDomainOffMem	Input	Tells the outside world and Access Port that the memory power domain is off	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
PsoDomainOffDebug	Input	Tells the outside world and Access Port that the debug power domain is off	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
G0ProcCLK	Input	Clock signal from PCM for core power domain; normally wired to CLK except during core domain power up/down, when PCM module controls it	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}
G0ProcBCLK	Input	Asynchronous bridge clock from PCM for core power domain; normally wired to BCLK except during core domain power up/down, when PCM module controls it	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, async bridge configured
G0DebugCLK	Input	Clock signal from PCM for debug power domain; normally wired to CLK except during debug domain power up/down, when PCM module controls it	PSODomains = core_debug_mem, PSOCoreRetention = {none, partial, full}, OCD or TRAX configured
PCMReset	Input	Reset input for all local PCM logic.	Requires PSO

11. Processor Interface (PIF) Main Bus

Xtensa processors perform traditional main memory and system accesses on a Processor Interface (PIF) port. The PIF is a configurable, high-performance, multi-master interface definition that is easily bridged to third party buses and interconnects. The full PIF protocol is described in the *Xtensa Processor Interface Protocol Reference Manual*.

Cadence also provides standard bus bridges to AHB-Lite and AXI. These bridges can be used to easily integrate an Xtensa processor into an existing system, or as a starting point for an alternate bus bridge development. The AHB-Lite or AXI bus can run at the same speed, slower, or faster than the processor. If the bus is running slower, two modes are supported, one a low-latency synchronous ratio mode, the other a fully asynchronous mode. If the bus is running faster than the processor, then it uses a fully asynchronous mode. Two modes are supported to run the bus at a slower speed, either a low-latency synchronous ratio mode or a fully asynchronous mode.

A PIF master port is used for outbound requests. The master port serves the following types of requests:

- All instruction and data cache misses, and castouts are serviced through the outbound PIF.
- Uncached and cache-bypass requests.
- Atomic operations.

Our processors can also be configured to have a slave port to handle inbound requests. These requests can access the processors local data and instruction memories. The slave port is useful for:

- Initializing instruction and data memories.
- Efficiently moving blocks of runtime data in and out of local memories. Data movement controlled through DMA accesses can be more efficient than processor directed movement, but requires greater software control.

11.1 System Configurations

Figure 11–28 illustrates two different ways that a system can be organized to connect Xtensa processors to memories, peripherals, and other processors through the PIF. On the left side of Figure 11–28, a traditional interconnection network is shown with a 3rd party bus used as the system backbone. If AHB-Lite or AXI are selected as the primary interface, then Cadence adds bridging logic to convert the processor's native PIF interface to AHB-Lite or AXI to easily integrate Xtensa processors into such systems. On the right side of Figure 11–28, the system is partitioned into two parts:

1. A high-performance interconnection network for fast access to high-speed memories and improved inter-processor communication, and
2. A peripheral and device network. The PIF protocol is an excellent choice for building a high-performance interconnection network.

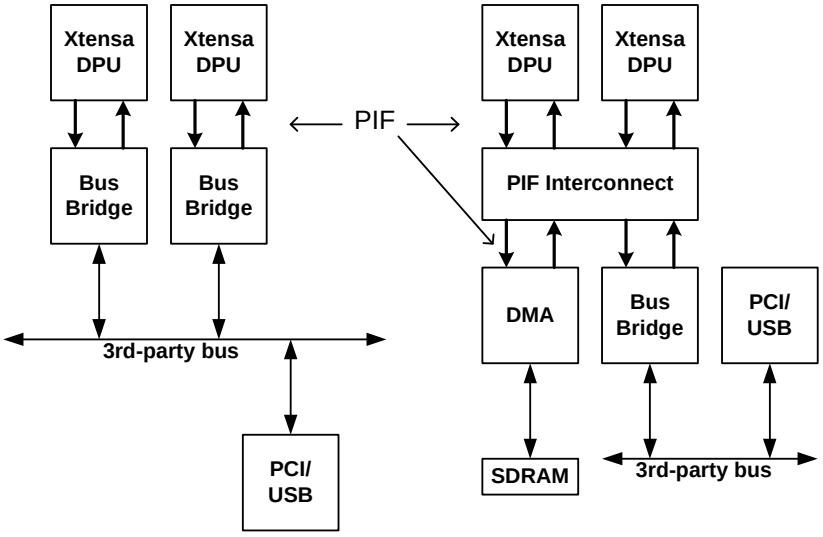


Figure 11–28. Examples of System Bus Interfaces Linked to the PIF Interface

Figure 11–29 shows how another example system that can be constructed using multiple Xtensa processors connected through their outbound and inbound PIFs.

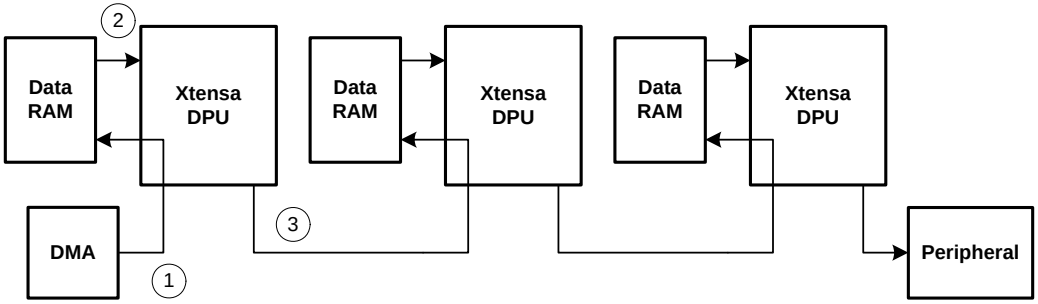


Figure 11–29. Multiple Processor System Using PIF to Interconnect Between Processors

In this example, multiple Xtensa processors are chained together to process a data stream. Each processor's local memory is used as intermediate FIFO buffers for the data stream. At point 1 in the diagram, a DMA engine writes into the first processor's local memory using its inbound-PIF capability. At point 2, the processor reads from its

local data RAM and operates on that data. At point 3, the first processor writes the data resulting from the computation into the next processor's local data RAM through its inbound-PIF channel and the cycle repeats down the processor chain.

11.2 Xtensa LX7 Processor Interface (PIF) Features

This section provides a general description of PIF 4.0 features, and how they relate to the Xtensa processor.

11.2.1 Synchronous and Pipelined Operation

CLK is the Xtensa processor core's clock input. PCLK is an internal processor signal derived from CLK and represents a typical leaf node of the non-gated clock distribution tree within the core. (Refer to Section 34.3 "PCLK, the Processor Reference Clock" on page 605 for further explanation.) Input signals are sampled on the rising edge of PCLK, with some input delay and setup time requirement. (See Figure 34–190 on page 606). Output signals are driven directly from registers on the rising edge of PCLK, with only clock-to-Q and routing output delay. This clocking scheme imposes a 1-cycle pipeline delay between every acknowledgment and response.

11.2.2 Unidirectional PIF Signals

All PIF signals are unidirectional.

11.2.3 Single Data Transactions

Read and write transactions smaller than or equal to the data-bus width require only one data-transfer cycle and are called single data transactions. Byte enables indicate which bytes should be read or written. The PIF can be configured with an optional write-response handshake mechanism to aid in reporting of transaction errors and for multi-processor synchronization. The processor issues single data transaction requests for uncached and cache-bypass requests, and for cache misses when a cache line fits within the configured PIF data-bus width.

11.2.4 Block Transactions

Block-read and -write transactions allow block data transfers that use multiple full-width PIF transaction cycles. The processor issues block requests for cache misses and castouts when the cache line is larger than the configured PIF data-bus width. Block requests are also issued for uncached read requests when the load width is wider than the PIF width. For example, a 128-bit designer-defined load translates into a 4-transfer block-read request for a 32-bit PIF. The processor issues store transactions that are wid-

er than the PIF as a series of back-to-back, single-data stores rather than as a block-store transaction. Block sizes of 2, 4, 8, and 16 transfers are allowed. The PIF can be configured with an optional write-response handshake mechanism to aid in reporting of transaction errors and for multi-processor synchronization.

11.2.5 Critical Word First Option

The Xtensa processor can be configured to issue block-read transactions as Critical Word First Transactions. Any block-read can be issued such that a specified PIF width of the block will arrive first, with the remaining PIF widths arriving in sequential order and wrapping around to the beginning of the block e.g. a block-read of 8 PIF widths could arrive as 5, 6, 7, 0, 1, 2, 3, 4. Requesting the most critical data first can allow the processor to restart more quickly after a cache miss.

11.2.6 Arbitrary Byte Enable Option

The Xtensa processor can be configured to use Arbitrary Byte Enables on write requests. This is useful when Arbitrary Byte Enable TIE is used that generates stores that have arbitrary byte enable patterns.

Note: Will only generate Arbitrary Byte Enables on write single transactions.

11.2.7 Read Conditional Write Transaction

Read-conditional-write transactions perform an atomic update of a memory location and are used to implement synchronization primitives for inter-processor or inter-process communication by providing mutually-exclusive access to data (for example, locks).

Note: These operations only occur if the Xtensa LX7 Conditional Store Synchronization Instruction is configured.

11.2.8 Multiple Outstanding Requests

Multiple transaction requests may be outstanding on the PIF. Transaction responses may be returned out of order and dynamic flow-control mechanisms are available to control the number of requests that are outstanding at any time. Six bits of transaction ID are available for each transaction request and these bits can be used by the requester in any fashion without restriction.

Note: A system need not handle multiple transactions. The number of outstanding transactions can be kept to one by not asserting the PIF ready lines.

11.2.9 Flexible Arbitration for System Bus

The processor interface protocol does not dictate a specific arbitration protocol and is easily adaptable to a variety of schemes through an external agent. Priority bits are defined for every transaction and can be used for arbitrating transaction requests from multiple sources.

11.2.10 Full Handshake-Based Protocol

A PIF master begins a transaction by issuing a valid *request* assertion, and a PIF slave indicates that it is ready to accept requests by asserting a *request ready* signal. A request transfer completes when valid request and request-ready signals are both asserted during the same cycle. The flow of each transfer within a block transaction can be controlled with this handshake mechanism.

Similarly, a PIF slave completes a transaction by issuing a valid *response* to the PIF master's request. The master accepts the response by asserting a *response ready* signal. A response transfer completes when valid response and response-ready signals are both asserted during the same cycle. The flow of each transfer within a block response can be controlled with this handshake mechanism.

These per-transfer handshake mechanisms permit more flexible component design and arbitration schemes. A PIF master can throttle its outbound request stream and its incoming response stream to match data bandwidth to available buffering. Similarly, a PIF slave can throttle its outbound response stream and its incoming request stream to match data bandwidth to available buffering.

11.2.11 Interleaved Responses

The individual PIF response transfers to different PIF-Block Read requests, as well as responses to PIF-Single Read and PIF Write requests, can all be interleaved with each other on the response bus. The response that a specific transfer is associated with is identified by the response transaction ID.

11.2.12 Narrow-Width Peripheral/Memory Support

There is no special support for I/O transactions to devices narrower than the PIF data buses. Xtensa processor cores align loads and stores within the appropriate data bus.

Consequently, a byte-wide peripheral device requires an 8:1 data mux to receive data from any byte lane of a 64-bit PIF data bus during write transactions, and the peripheral device must drive its byte-wide data to the specified byte lane during PIF read transactions.

A bus interface or external bus bridge that is aware of the external peripheral/device requirements (endianness and data width) is required to align data between the PIF and an external bus.

11.2.13 Write Buffer

Xtensa processors implement a configurable-depth write buffer to queue store requests and cache-line castouts. The depth of the write buffer can be tuned to the PIF bandwidth available to a configured processor during the development process. The ordering of data loads and writes can also be configured to decrease PIF read latency at the expense of a more relaxed memory ordering.

11.2.14 Inbound-PIF Option

A slave PIF may be configured to handle inbound requests to the Xtensa processor's local memories, which include instruction and data RAMs.

11.2.15 Request Attribute Option

The PIF can be configured to have an additional Request Attribute output for a PIF master (POReqAttribute[11:0]) and input for a PIF slave (PIReqAttribute[11:0]). The attribute indicates various characteristics of the request (cacheable or non cacheable, instruction or data, etc.) that are useful for implementing system level features, such as a second level cache. The bottom 4 bits of the outbound POReqAttribute can be user defined, and can be driven by user-defined loads and stores.

11.2.16 No-PIF Configuration

An Xtensa processor can be configured without a PIF for SOC designs that require the lowest possible gate count or for deeply embedded uses that do not require access to a global memory structure. Xtensa processors configured without a PIF must operate entirely from local memory and do not support cache memory.

11.3 PIF Configuration Options

The following configuration options are available:

- No-PIF option:
  The Xtensa processor interface is optional.

  Note: Xtensa processors configured with caches must be configured with a PIF to service cache refill and castout operations.

- Processor Data Bus Width:
  All request and response data buses in a PIF must be the same width and can be 32, 64, or 128 bits wide.
- Cache Block-Transfer Size:
  The bus master supports multi-cycle block-read transactions for cache refills and multi-cycle block-write transactions for castouts. Xtensa processor cores can be configured with different cache-line sizes (and block read/write sizes) for instruction or data caches. A cache refill or castout may require 1, 2, 4, 8, or 16 data-bus-width transfers, depending on the PIF data bus width and the cache line size.
- Block-Read Request:
  Block-read requests are issued for cached line refills when the cache line is larger than a data-bus width. The smallest block size that encompasses an entire cache line is used for each cache-miss request. The instruction and data cache may have different block sizes. Block-read requests are also issued for uncached instruction fetch or uncached loads that are larger than the configured bus width. For example, a 128-bit, designer-defined load will translate into a 4-transfer block-read request on a 32-bit bus or a 2-transfer block-read request on a 64-bit bus.
- Block-Write Request:
  Xtensa processors only issue block-write requests for dirty-line castouts when they are configured with a write-back cache and their cache line is larger than the bus data width.
- Write-Buffer Depth:
  A write buffer of 1, 2, 4, 8, 16 or 32 entries is used to mask PIF latency during large bursts of writes, data-cache dirty-line castouts, and register spills. This buffer must be deep enough to store at least one cache line.
- Prioritize Load before Store:
  Data reads and writes can be configured to appear in program order, or with a relaxed memory ordering that permits data reads to bypass writes to reduce read latency.
- Write Responses:
  Xtensa processors can be configured to count write responses for memory-ordering or synchronization purposes. When the write-response configuration option is selected, 16 unique IDs are allocated to write requests, and no more than 16 write requests can be outstanding at any time. Store-release (S32RI), memory-wait (MEMW), external-wait (EXTW), and exception wait (EXCW) instructions will wait for all pending write responses to return before those instructions commit. The Xtensa processor ignores any write response with a response ID set to a value that does not correspond to an outstanding write request. Consequently, only responses with a matching ID can complete a write transaction waiting on a response.

  When the Xtensa processor is not configured to count write responses, all write requests will be issued with the same ID, and S32RI, MEMW, EXTW, EXCW instructions will not wait for write responses before committing. A PIF slave can still send

write responses to an Xtensa processor, even if write responses were not configured for that processor. An Xtensa processor configured without write responses will accept and ignore these responses. Designers may want to design their bridges to always issue write responses for compatibility across multiple processor configurations.

- Write-Bus-Error Interrupt:
  Address and data bus errors reported in write responses do not generate precise exceptions, because the instruction that initiated the store operation would generally have committed in the processor's pipeline by the time a write response could be returned to the processor. Instead, bus errors reported in write responses generate an internal interrupt. This interrupt is maskable by both the interrupt level and interrupt-enable register, and may be configured as a high-priority or level-1 interrupt. The exception virtual address is not available for write-bus errors.
- Read-Conditional-Write Transaction:
  The read-conditional-write transaction implements the optional store-32-bit-compare-conditional instruction (`S32C1I`). The `S32C1I` instruction is a conditional store instruction intended for updating synchronization variables in memory.

  Depending on the state of the ATOMCTL register, the read conditional write (RCW) PIF request may be generated by the Xtensa processor for each S32C1I instruction. If the PIF slave or associated external memory controller does not support conditional writes on the PIF, then its behavior can be changed in the ATOMCTL register to either take an exception or handle the S32C1I internally.

  Note: Xtensa configurations that include an MMU-with-TLB require the Store Compare Conditional (S32C1I) instruction. The Cadence Linux port uses this instruction to provide synchronization among multiple tasks or exception handlers running on a single processor, including synchronization among user-level tasks where disabling interrupts is a privileged operation.
- Inbound-PIF Request:
  An Xtensa core may optionally accept inbound requests on a separate PIF slave interface that accesses the local instruction RAMs or data RAMs. The processor must have a PIF for the inbound-PIF Request option to be enabled. The processor must also have at least one instruction RAM or data RAM to be compatible with this option. Inbound-PIF requests are configured separately for each type of configured local memory.
- Inbound-PIF Request and Response Buffer Sizes:
  The inbound PIF implementation includes three buffers: the request control buffer, the request data buffer, and the response buffer. The depth of all three of these buffers is the same depth and is configurable. Every inbound-PIF transaction request occupies one entry in the request control buffer. Write requests occupy one entry in the request data buffer for each transfer required by the request. Data requests occupy one entry in the request data buffer. Read-conditional-write requests occupy two entries in the request data buffer.

The processor will control the flow of inbound-PIF requests if the request control buffer or the request data buffer is full by driving the POReqRdy signal low. The processor pipelines block-write requests so it can accept block-write requests even when it does not have enough empty entries in the request data buffer to hold the entire request block. Consequently, the processor may exert flow control on the individual transactions in the request block on the PIF. However, the processor issues each transaction of an inbound-PIF request to a local memory as soon as it's received.

To optimally size the request buffer depth, the designer must consider the size of blocks that will be issued, the rate at which requests are issued, and the priority of inbound-PIF transaction requests relative to the processor's own requests for the same memory. All inbound-PIF transaction responses go through the response buffer.

Note: PIF request and response data widths are always equal.

- Critical Word First Option:
  The PIF can optionally be configured to make cache line block-requests Critical Word First. Note that the early restart option must also be configured in order to select this option.
- PIF Arbitrary Byte Enables:
  The PIF can optionally be configured to allow write requests to use arbitrary byte enables on the outbound PIF write requests. Note that this helps when Arbitrary Byte Enable TIE is used because stores that would otherwise be broken up into several stores can go out as a single store.

  Note: The processor outbound PIF will only ever generate PIF Arbitrary Byte Enables on single write requests.
- Request Attribute:
  The PIF can optionally be configured to have the Request Attribute Option, which provides information about the type of PIF request being made, such as whether it is a cacheable or uncacheable request, a data or instruction fetch request, etc. This information is intended to be useful in the memory system implementation. See Section 13.2.3 for additional details.

12. Inbound Processor Interface (PIF)

This chapter describes the PIF 4.0 protocol as implemented by the Xtensa processor PIF inbound port. The Xtensa PIF implementation fully complies with the PIF protocol, but it does not use all of the features of the full PIF protocol. Future PIF implementations on Xtensa processors may implement other features of the PIF protocol. Cadence may also add new transactions to the PIF protocol in the future.

12.1 Inbound-PIF Transactions

The Xtensa processor can process inbound read and write requests from an external agent through an inbound-PIF channel. This inbound-PIF channel is a physical channel (wires), separate from the processor's outbound PIF port that services cache misses and other processor requests. The processor's inbound-PIF request option is available when the outbound PIF option is enabled and at least one instruction RAM, or data RAM are configured. Inbound-PIF operations are not allowed to addresses allocated to local ROM's, or caches, or the processor's XLMI port. The processor has a buffer, of configurable depth, which is used to both pipeline multiple requests as well as relieve bus contention by buffering requests within the processor.

All inbound-PIF request addresses must be physical addresses relative to the processor accepting the request. That is, inbound-PIF request addresses are not translated, regardless of whether or not the processor has a TLB. Inbound-PIF requests completely bypass the TLB and are not subject to TLB protection.

From the perspective of a memory or device on the instruction- or data-RAM interface, inbound-PIF requests are indistinguishable from requests generated by instructions executed by the local processor. Inbound-PIF requests to an interface with a busy handshake signal may stall the processor if that interface is busy during the execution of the inbound-PIF request. In addition, inbound-PIF requests are not speculative, i.e they will all retire.

12.2 Support of PIF Protocol 4.0 Options

The Xtensa LX7 processor implements PIF Protocol 4.0, which is compatible with Versions 3.0, 3.1, and 3.2 of the PIF protocol, but also includes optional support of PIF Arbitrary Byte Enables, Request Attribute, and Critical Word First. To enhance the cache system performance, these new features are supported optionally in Xtensa LX7 processor for outbound-PIF requests. Given that inbound-PIF request is typically used for initializing instruction RAM and moving data in/out data RAM. These new optional features are selectively supported on inbound-PIF port.

- Write with arbitrary byte enables: Supported by default on the inbound-PIF port.
- Request attribute: The Xtensa processor does not use the information in the inbound PIReqAttribute[11:0] input signal if the request attribute option is configured.
- Critical-word-first inbound-PIF requests: The Xtensa processor does not support inbound critical-word-first requests.

12.3 Inbound-PIF Transaction Details

The Xtensa processor services inbound-PIF requests for data reads/writes. Each data read/write request can be a single request or a block request controlled by the PIReqCntl bits. The Xtensa processor also services single-word RCW requests if the Conditional Store Synchronization Instruction option is selected.

12.3.1 Inbound-PIF Write Requests

Figure 12–30 shows the ideal timing of two back-to-back inbound-PIF block write requests. In this example, each of the block writes consist of four transfers. Figure 12–30 shows the following:

Latency

- There is a minimum 2-cycle latency for each write to the local memory (cycle 2 to cycle 4 for the first write) from the time it is accepted on the PIF.
- Assuming that the local memory has a 1-cycle latency, there is a 2-cycle latency from the last successful write of a block write request to the write response being asserted on the PIF (cycle 7 to cycle 9 for the first block write). Each additional cycle of latency in the local memory adds to the response latency.

Bandwidth

An ideal timing diagram for two block write requests to the local memory is shown Figure 12–30. These two consecutive block write requests can be serviced by the Xtensa core in a back-to-back manner. Note that the Xtensa core's RAM-to-PIF width ratio, internal buffer status and busy conditions, etc. may change this timing relationship.

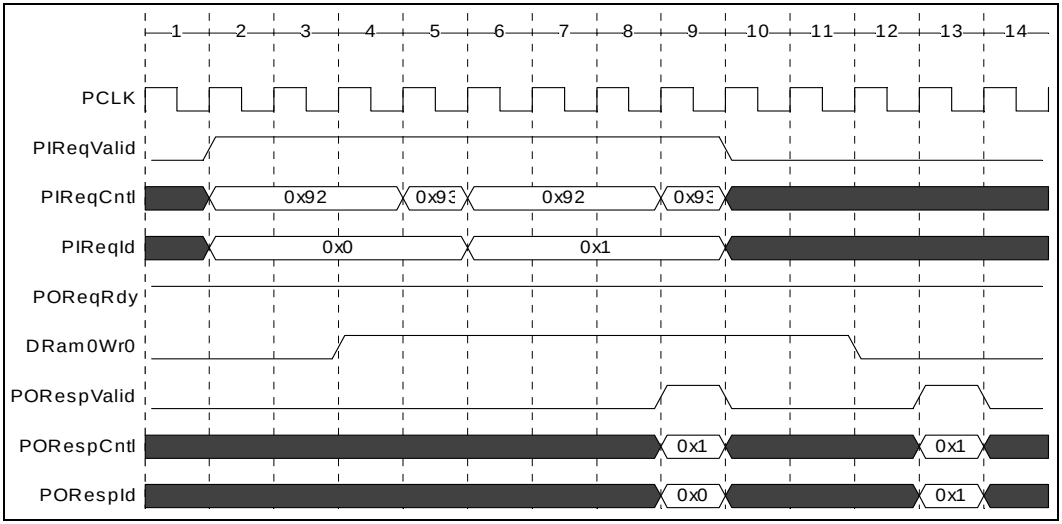


Figure 12–30. Inbound-PIF Write Requests

12.3.2 Inbound-PIF Read Requests

Figure 12–31 shows the ideal timing of back-to-back inbound-PIF block read requests. In this example, each block read has a size of four data transfers. Figure 12–31 shows the following:

Latency

- There is a minimum 2-cycle latency from acceptance of the PIF request (cycle 2) to the request being issued on the local memory (cycle 4).
- Assuming that the local memory has a 1-cycle latency, there is an additional 1-cycle of latency from the local memory response to the inbound-PIF response appearing on the PIF (cycle 5 to cycle 6). Each additional cycle of latency in the local memory adds to the response latency.

Bandwidth

An ideal timing diagram for two block read requests to the local memory is shown in Figure 12–31. These two consecutive block read requests can be serviced by the Xtensa core in a back-to-back manner. Note that Xtensa core's RAM-to-PIF width ratio, internal buffer status and busy conditions, etc. may change this timing relationship.

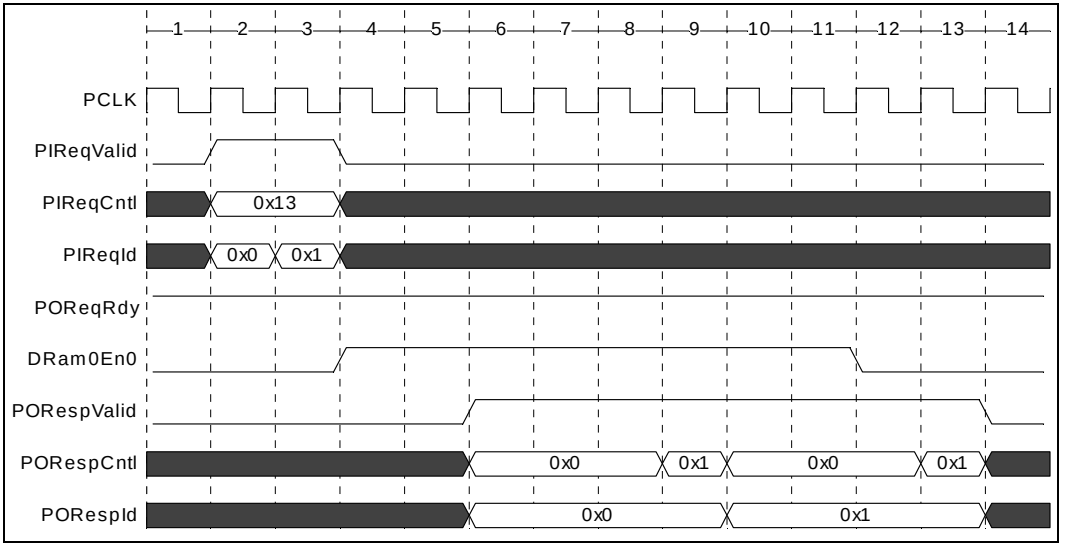


Figure 12–31. Inbound-PIF Read Requests

12.3.3 Inbound-PIF Read Conditional Write Requests

The processor accepts inbound-PIF single-word read-conditional-write (RCW) requests when the Conditional Store Synchronization Instruction option is selected. RCW requests are 2-transfer requests. RCW performs a read as its first transfer. The read latency is the sum of memory latency and the number of memory busy cycles. After the read data comes back, data comparison is performed to determine whether a subsequent write is permitted. This conditional write will be issued to memory only if the comparison succeeds. Then the RCW transaction completes. The local memory Lock interface is asserted during the entire RCW operation. (If the processor does not have the Conditional Store Synchronization Instruction option, then an inbound-PIF-initiated RCW request generates a bus error, documented in Section 12.4.7.)

12.3.4 Inbound-PIF Read Modify Write Operations

If the inbound PIF accessible data RAMs are configured with the wide ECC/Parity option, an inbound PIF partial write will trigger a read-modify-write operation if any word in this request will only be updated partially. For example, write only one byte in a word will trigger the read-modify-write sequence. The read-modify-write operation takes multiple cycles to do the following steps:

- Read: inbound PIF reads the original data from the data RAM
- Modify: inbound PIF modifies the write data by merging the portion that will be updated with the portion that will remains unchanged.

- Write: inbound PIF writes the modified data into the data RAM.

The multi-cycle inbound PIF read-modify-write sequence is carried out in an atomic manner with regard to the other agents, such as LoadStore units, which may attempt to update the datum at the same time, thus may causing data hazard. A mutex mechanism is deployed to enforce that either the LoadStore units or the inbound PIF can do a Read-Modify-Write at a time when there is potential data hazard.

Note: The competition for the read-modify-write mutex between the other agent and the inbound PIF may impact the performance. Because before one can proceed, it may have to wait for the other to release the mutex. In some cases, even high-priority inbound PIF will have to wait because of the mutex. Such a performance impact is implementation dependent. Therefore, for performance critical applications, it is suggested to use partial writes with caution and separate the inbound PIF accesses and the other agent's accesses to different data RAMs.

12.4 Inbound PIF Implementation Details

This section describes inbound-PIF implementations such as requests, the "PIF arbitrary byte enables" option, data width conversion, and parity or ECC errors.

12.4.1 Inbound-PIF Request Buffer

The inbound-PIF request buffer holds inbound-PIF block-read and -write requests, and has a configurable depth. It is possible to issue inbound-PIF block requests that exceed the buffering this buffering, in which case the processor may flow control requests on the PIF. The designer should consider the bandwidth versus area tradeoff of buffering requests.

A separate response buffer to hold responses is also present, with the same depth as the inbound-PIF request buffer.

12.4.2 Synchronization and Arbitration of Inbound-PIF Requests and Processor Requests

Inbound-PIF requests have higher priority than the processor's instruction-fetch operations when both operations attempt to access the same instruction RAM simultaneously. The processor can continue without stalls if it is executing code located in a different instruction RAM or from the instruction cache during inbound-PIF operations.

The RunStall pin can be driven high to stop the processor pipeline while inbound-PIF operations are used to initialize the instruction RAM. While it is possible to continue fetching instructions from an instruction RAM while inbound-PIF operations also reference the RAM.

When inbound-PIF requests target the processor's local data RAM, the assigned inbound-PIF request priority, PIReqPriority, is used to arbitrate between the local processor's load and store operations and inbound-PIF requests for access to the local RAMs. A priority of 0x3 gives inbound-PIF the preemptive access to the local memory and stalls processor-generated load and store requests to that same memory bank. Loads and stores to other memories or memory banks will be allowed to progress without stalls. A priority of 0x0 to 0x2 will result in the inbound-PIF requests getting access to the local memory when they are free. That is, an inbound-PIF request will be issued to a local memory whenever that memory is not handling processor-generated load or store operations. In addition, inbound-PIF requests are guaranteed to make forward progress by stalling the processor's pipeline if the inbound-PIF request has been blocked for several cycles by processor-generated operations.

The inbound-PIF access to data RAMs is independent of the Xtensa pipeline. Inbound PIF requests and core load/store instructions can access data RAMs in parallel, if they are targeted to different data RAMs (if multiple data RAMs are configured) or to different data RAM banks (if multiple banks are configured). For applications that need to maintain certain data dependency, or accessing into the same word between inbound PIF requests and load/store instructions, the system designer must take special care in synchronizing the two with the help of software or hardware synchronization techniques, such as, Read-Conditional-Write transactions and the S32C1I instruction.

Back-to-back read/write requests (except RCW) can be serviced by inbound-PIF with no overhead. This provides the best use of the available PIF and local RAM bandwidth. For example, when the PIF width matches the data RAM width, in an ideal setup, a continuous stream of high-priority single-write requests can utilize 100% bandwidth because of the back-to-back RAM accesses.

If there is no contention, even low-priority inbound-PIF requests to data RAMs may be serviced with the full bandwidth. If there are conflicts between inbound-PIF requests and the load/store instructions, the Xtensa core can resolve the conflict and maintain the performance of inbound-PIF requests close to an allotted bandwidth.

Table 12–54 summarizes the bandwidth allotted to inbound PIF requests to data RAM as a function of the request priority, assuming that data RAM width matches PIF data width. Note that the actual percentage of RAM bandwidth achieved is impacted by static and dynamic factors, such as the RAM-to-PIF width ratio, or how fast the PIF responses can be dispatched, etc. The upper bound of the achievable data RAM bandwidth is the lower of the PIF and data RAM bandwidth.

Configurations with two load/store units and the C-Box option have additional arbitration effects on the available bandwidth as described in Section 18.9.3 “Inbound PIF to Data RAM Transactions” on page 321.

Note: Use high-priority inbound PIF (PIReqPriority=2’b11) accesses with caution as issuing a long sequence of back-to-back high-priority inbound PIF accesses may stall the Xtensa processor indefinitely if they both compete for the same data RAM bank.

Table 12–54. Inbound Request Priority

PIReqPriority [1:0]	% of Data RAM Bandwidth Allocated to Inbound PIF
11	100%
10	Approximately 25%
01	Approximately 25%
00	Approximately 25%

Note: Configurations with wide ECC/Parity option may utilize mutex to protect read-modify-write operation. The effect of mutex can impact the inbound PIF performance as described in Section 12.3.4 “Inbound-PIF Read Modify Write Operations” on page 194, thus changing the effective bandwidth.

12.4.3 Inbound-PIF Block-Request Addressing

Inbound-PIF bock-request addresses must be aligned to the size of the complete transfer. Neither critical-word-first nor unaligned addressing is allowed. That is, the low-order log2 (data bus width) + log2 (block size) bits of the address must always be 0x0.

Block read requests not conforming to this address alignment requirement will receive address bus error responses. Block write requests not conforming to this address requirement violate PIF protocol and have undefined behavior.

12.4.4 Inbound-PIF Arbitrary Byte Enables Option

If the "PIF arbitrary byte enables" option is selected, the Xtensa processor may issue single writes on the outbound PIF, in which any or all of the PIF byte lanes may be enabled. For more information, refer to the *Xtensa Processor Interface Protocol Reference Manual*.

Whether or not the PIF arbitrary byte enables option is configured for an Xtensa processor, the inbound PIF interface always accepts single and block write transactions with arbitrary byte enables, as defined in the PIF protocol.

For single writes on the inbound PIF, the first enabled Byte Enable in the transaction must be for a byte lane at an address greater or equal to the write address. For example, for writes to a data RAM in a 128-bit PIF configuration, any of 16 bytes may be enabled. Also, a write transaction might not enable any bytes, meaning it has no effect. For example, address 0x01000004 in a little-endian configuration would map to the fifth byte lane, with ReqDataBE[15:0]= 0x0010. Thus, for this address in a data RAM, a valid write Byte Enable may be 0x0000, 0x1100 or 0x10f0, while Byte Enable 0x100f is invalid for this address.

Note: The Xtensa inbound PIF behavior is undefined for single writes with any byte enabled below the write address, and may lead to unpredictable results. In this case, the Xtensa processor may or may not return a PIF error response for the transaction.

For block writes, the starting address must be aligned to the block size and not change throughout the transaction. For data RAM accesses, any combination of byte enables is legal in any transfer of the block write. Block writes which do not enable any bytes on the PIF are also allowed.

The instruction RAM must be accessed in a 32-bit word unit. Each 32-bit word must have the contiguous four bytes all enabled or all disabled. For single and block writes to the instruction RAM, each 32-bit word can be enabled or disabled arbitrarily. Not enabling/disabling bytes for instruction RAM writes in the granularity of 32-bit words may lead to undefined write behavior. Those accesses that have all 0 byte enables for the whole instruction RAM width may be suppressed on the instruction RAM interface for performance reasons.

For single reads, the PIF protocol requires that the address must be aligned to the data width that you are reading and all bytes must be enabled for the data width.

For block reads, the PIF protocol requires that the starting address must be aligned to the block size and all bytes must be enabled.

Note: The Xtensa processor core only supports 32-bit RCW on the inbound-PIF port. The inbound-PIF RCW requests must have addresses aligned to 32-bit words. The four contiguous bytes must be all enabled. Not complying with this requirement may lead to unpredictable results.

12.4.5 Inbound-PIF Request Flow Control, Ordering and Multiple Outstanding Transactions

The request buffering depth is a configurable parameter of the processor. The processor uses the POReqRdy signal for flow control if accepting any new requests might overflow the configured request buffer. This buffering allows the Xtensa processor to accept several inbound requests from the PIF before setting POReqRdy low. If the inbound-PIF re-

quest buffer is empty and the local memory is not being used by load/store operations in the pipeline, then the inbound-PIF request buffers are generally bypassed and PIF requests will generally not require flow control.

A response buffer, with the same configured depth as the request buffer, is available to buffer responses in the event that the PIF is unavailable as indicated when PIRespRdy is not asserted. Note that the Xtensa processor responds to all inbound requests in the order they are received.

Multiple outstanding transactions are possible on an inbound-PIF interface depending on the buffer depth, type, and size of Inbound-PIF requests, and what request is currently being processed.

12.4.6 Inbound-PIF Data Width Conversion

The PIF, data memory, and instruction memory widths can all be set independently, with the restriction that the PIF width cannot be greater than the data memory width. When the PIF width is not equal to the target memory width, some width conversion must occur which has an effect upon the bandwidth and latency of the inbound-PIF operation.

Data Memory

If the data-memory width is wider than the PIF width, multiple PIF write transfers (within a block write) will be coalesced before the data memory is written to optimize data-memory bandwidth. Reads are performed at the data memory width and every data-memory read that is caused by a block read request may require multiple PIF read response transfers.

Instruction Memory

Optimization of instruction-memory bandwidth during inbound-PIF operations is generally not required because it is difficult to initialize an instruction RAM while simultaneously executing from it. Therefore, if the instruction memory is wider than the PIF, inbound-PIF operations are performed at the PIF width. When the instruction memory is narrower than the PIF width, each PIF transfer may require multiple instruction-memory accesses.

12.4.7 Errors on Inbound-PIF Requests

Errors on inbound-PIF read requests are encoded in the response. If write responses are configured, errors generated by inbound-PIF write requests are also encoded in the response. If write responses are not configured, then errors on inbound-PIF write requests are not communicated back to the external agent.

The processor detects four types of errors for inbound-PIF requests. The first three errors return a single-cycle address error response. The last error will return a data error response. The types of errors are:

- An illegal address error: An address error is generated for inbound-PIF requests made to addresses outside the address space of a local instruction or data RAM. Such an address error is generated for reads and for writes when the write response option is configured. An address error is also generated for inbound-PIF block read requests to unaligned or critical-word-first addresses.
- An unknown or unsupported command: Unknown commands may arise in incorrectly designed hardware or from PIF protocol incompatibilities caused by combining components designed to interface to different Xtensa processor models. If the Conditional Store Synchronization Instruction option is not selected, read-conditional-write requests will receive a bus error.

  Important Note: Even when a processor does not have write responses, an unknown or unsupported command will return an address error response.
- If inbound-PIF byte enables for instruction RAMs are not enabled or disabled in the unit of 32-bit words for single read, single write or block write, the Xtensa core will respond with a PIF address error for read and for write if the write response option is configured. The behavior of instruction RAM writes with such invalid byte enables is undefined and may lead to unpredictable results. For inbound-PIF block reads, not enabling all bytes for the PIF width violates PIF protocol and its behavior is undefined.

  Note: The Xtensa inbound PIF behavior is undefined for single writes with any byte enabled below the write address, as described in Section 12.4.4. In this case, the Xtensa processor may or may not return a PIF error response for the transaction.
- Inbound-PIF read requests to RAMs with memory error detection configured and enabled will return good data. If an uncorrectable error is detected on the read, a data error response is generated for the bad data.
- The inbound-PIF read-modify-write operation (when the wide ECC/Parity option is configured) gets uncorrectable error during data RAM read: the read-modify-write will skip the write operation for the affected word. If write response option is configured, the inbound PIF responses with a data error for the write. If write response option is not configured, the inbound PIF will just leave the error in the data RAM allowing it to be detected by the processor when it is accessed from the pipeline.

Errors on inbound-PIF requests are only reported back to the external agent (encoded in `PORespCntl`), not to the processor servicing the inbound-PIF request.

12.4.8 Configurations with Memory Parity or ECC Option

Errors are automatically detected for inbound PIF reads from locations with a parity or ECC errors. Memory errors are reported as data bus errors in the response. In ECC configurations, read requests with errors that are corrected on the fly will not report a data bus error in the read response or assert the ErrorCorrected output pin.

Parity and ECC bits are automatically generated for inbound PIF write requests.

12.4.9 Inbound-PIF Requests and Power Shut Off (PSO)

The inbound-PIF requests go through the Xtensa processor to access the local memories. To service inbound-PIF requests, both core and memory power domains must be on. It is the responsibility of external PIF masters to stop requesting and ascertain that responses for all outstanding requests have been received before the SoC can allow the Xtensa core and memory to shut power off.

13. Outbound Processor Interface (PIF)

This chapter describes the PIF 4.0 protocol as implemented by the Xtensa processor PIF outbound port. The Xtensa PIF implementation fully complies with the PIF protocol, but it does not implement all of the features of the full PIF protocol. Future PIF implementations on Xtensa processors may implement other features of the PIF protocol. Cadence may also add new transactions to the PIF protocol in the future.

Note: For system designs using Xtensa processor configurations with a PIF, physical-memory accesses not matching the cache tags or within configured local-memory address ranges are automatically issued on the PIF. Consequently, system designs using an Xtensa processor with a PIF must detect PIF accesses to unimplemented physical memory and report these accesses as errors to properly complete the PIF transaction.

13.1 Outbound-PIF Transactions

The Xtensa processor uses the outbound PIF port to read and write data. It reads uncached data as well as complete cache lines, and it writes uncached data as well as writing out complete cache lines of castout data back to memory. If certain configuration options are chosen the outbound port is used to do Read-Conditional-Write (RCW) atomic operations as well.

All outbound-PIF request addresses are physical addresses. In the case the processor has a TLB, the virtual addresses have been translated to physical addresses before being issued on the PIF.

The outbound PIF has an extensive amount of configurability which typically trade off performance for area.

13.2 Support of PIF Protocol 4.0 Options

The Xtensa LX7 processor implements PIF Protocol 4.0, which is compatible with Versions 3.0, 3.1, and 3.2 of the PIF protocol except for optional support of PIF Critical Word First, Arbitrary Byte Enable, and Request Attribute. To enhance the cache system performance, these new features are supported optionally in Xtensa LX7 processor for outbound-PIF requests. These options are described below.

13.2.1 Optional Critical-Word-First

The Xtensa processor can be configured to issue critical-word-first block read requests on the PIF for its cache line read requests. In the case that the processor can restart the pipeline as soon as the critical data is available, a critical-word-first block read will improve performance by allowing the pipeline to restart earlier on a cache miss.

If the Critical-Word-First Option is configured, then for block-reads the bottom bits of the PIF POReqAdrs will point at a specific PIF width transfer within an aligned cache line block, and the response must return this PIF width transfer first, and then the rest of the cache line in wraparound fashion. The outbound PIF will never issue block-writes critical-word-first.

If the Critical-Word-First Option is not configured, then the processor issues request addresses that are aligned to the size of the complete transfer. Consequently, block-request addresses are always aligned to the size of the block. That is, the low-order $\log_2$(data bus width) + $\log_2$(block size) bits of the address will always be zero.

13.2.2 Optional Arbitrary Byte Enables

The Xtensa processor can be configured to allow single writes with arbitrary byte enables. This is useful to reduce the number of PIF requests when the user has TIE store instructions that use Arbitrary Byte Enables, or when unaligned stores are used. The Xtensa store operations in the base instruction set always issue aligned addresses, and always with byte enables as defined in the *Xtensa PIF Protocol Reference Manual*.

If PIF Arbitrary Byte Enables are configured then the processor can use single write operations that have Arbitrary Byte Enable patterns whenever it is convenient to do so. The only restriction is that bottom bits of the POReqAdrs point at a byte that is lower than the first byte that has a non-zero Byte Enable.For instance, for a 128-bit wide PIF (16 Bytes), and a byte enable pattern of 0x51f8, the bottom four bits of the P0ReqAdrs can legally be 0x0, 0x1, 0x2, 0x3, but not 0x4 or higher. The processor will not issue block writes with Arbitrary Byte Enables.

If PIF Arbitrary Byte Enables are not configured, then the processor will only issue certain address and byte enable patterns on the PIF. Specifically, the byte enables must point at an aligned power of 2 bytes (1, 2, 4, 8, or 16-bytes), and the POReqAdrs will point at the byte corresponding to the first non-zero Byte Enable.

Note that if a processor is not configured with PIF Arbitrary Byte Enables, but a user-designed TIE store operation uses byte enable patterns not defined by the PIF Protocol, then a single store will be broken into multiple stores. For example, in a little endian configuration, a 128-bit store instruction to address `0x0` with a store byte disable of `0x1`

would nominally generate the byte-enable pattern `0xfffe`, which is an illegal byte-enable pattern on the PIF if Arbitrary Byte Enables are not configured. Consequently, the Xtensa processor breaks this store operation into four separate store transactions:

1. Address `0x1`, byte enable `0x2`
2. Address `0x2`, byte enable `0xc`
3. Address `0x4`, byte enable `0xf0`
4. Address `0x8`, byte enable `0xff00`

For a big-endian configuration, the store would also be broken into four separate store transactions with a different set of byte-enable patterns:

1. Address `0x0`, byte enable `0xff00`
2. Address `0x8`, byte enable `0xf0`
3. Address `0xa`, byte enable `0xc`
4. Address `0xe`, byte enable `0x2`

13.2.3 Optional Request Attribute

The Xtensa processor can optionally be configured to include the PIF 4.0 Request Attribute feature. The Request Attribute Option defines a 12-bit Request Attribute field (signals POReqAttribute[11:0] for a master PIF port, and PIReqAttribute[11:0] for a slave PIF port), that encodes optional sideband information with every PIF request.The 12-bits of the POReqAttribute field are used as follows for the Xtensa outbound PIF port:

- POReqAttribute[11:4]: The top 8 bits provide information about the type of request and the attribute of the memory it is requesting. For instance it says whether it is an instruction or data request, whether it is a cached request or uncached request, and whether it is a privileged request.This information is useful in implementing such things as a second level cache.
- POReqAttribute[3:0] These are user defined bits that can be set by user-defined TIE load and store operations.

Note that PIReqAttribute[11:0] is an input to the slave inbound-PIF port when the Request Attribute Option is defined, but this input is not used for any purpose by the Xtensa processor.

Pre- Defined Request Attribute Bits

Tables below show the pre-defined bits for the outbound POReqAttribute signal of the PIF 4.0 Request Attribute Option. Note that Per-Attribute bits are derived from the TLB cache attribute of the access.

Table 13–55. POReqAttribute PIF Master Pre-Defined Bits

Signal Name	Name (Short Name)	Description
POReqAttribute[11]	Reserved(R)	Reserved for future use
P0ReqAttribute[10]	Priviledge(P)	Privilege bit for processors with rings (full MMU). Set if CRING=0 for most instructions except for L32E/S32E in which case it is set if PS.RING=0.For PIF accesses not clearly attributable to a specific instruction, such as castouts or prefetches, the P bit is set to 0. Always set to 1 for processors without rings.
P0ReqAttribute[9]	Instruction(I)	set to 1 for an Instruction fetch PIF request. IPFL fetches are marked as Instruction fetches. Prefetches due to hardware instruction prefetch are also marked as instruction fetches.
POReqAttribute[8]	Write-Allocate(WA)	Request from a write-allocate memory region. Set Per-Attribute.
POReqAttribute[7]	Read-Allocate(RA)	Request from a read-allocate memory region. Set Per-Attribute.
P0ReqAttribute[6]	Writeback(WB)	Request from a writeback region. Set Per- Attribute.
POReqAttribute[5]	Cacheable (C)	Request from a cacheable region. Set Per- Attribute.
POReqAttribute[4]	Bufferable (B)	Request is bufferable. A bufferable request can receive a response from an intermediate point in the response network, a non-bufferable must receive the response from the final destination. Set Per- Attribute, but can be overridden by the instruction (e.g. S32RI, S32NB, S32C1I all set the bufferable bit to 0 - see Table 13–58,).

Certain bits of the Request Attribute depend on the Cache Attribute of the memory region they are accessing to set their value. These are shown for the Region Protection and Full MMU TLB options. The B-bit can be overridden by certain store instructions.

Table 13–56. Request Attributes as a Function of Region Protection Cache Attributes

Region Protection CA	Cache Attribute Name	WA-Bit	RA-Bit	W-Bit	C-Bit	B-bit
2	Bypass	0	0	0	0	0
6	Bypass bufferable	0	0	0	0	b
0	Cached no-allocate	0	0	0	1	b
1,3	Write-through	0	1	0	1	b
4	Writeback write-allocate	1	1	1	1	b
5	Writeback no-write-allocate	0	1	1	1	b

Note: "b" means normally 1, unless overridden and set to 0 by an instruction such as S32NB, L32AI, S32RI, S32C1I

Table 13–57. Request Attributes as a Function of Full MMU Cache Attributes

Full MMU CA	Cache Attribute Name	WA-Bit	RA-Bit	W-Bit	C-Bit	B-bit
0-3	Bypass	0	0	0	0	0
4-7	Writeback Write-allocate	1	1	1	1	b
8-11	Write-through	0	1	0	1	b

Note: "b" means normally 1, unless overridden and set to 0 by an instruction such as S32NB, L32AI, S32RI, S32C1I

PIF Request Attribute bits may also vary as a function of the type of access, or of the specific instruction. Table 13–58 describes this, and indicates the value of all bits for PIF accesses that are asynchronous to the instruction stream such as castouts and prefetches.

Table 13–58. Request Attributes as a Function of Instruction or PIF Access

Access Type	PIF Access	R	P	I	WA,RA, W,C	B	User
I-fetch	cached read, uncached read	0	p	1	CA	B	0
IPFL	cached read	0	p	1	CA	B	0
Load	cached read uncached read	0	p	0	CA	B	u
L32AI	cached read uncached read	0	p	0	CA	0	0
DPFL	cached read	0	p	0	CA	B	0
DPFR, DPFRO, DPFW, DPFWO	cached read	0	p	0	CA	B	0
auto-prefetch	cached read	0	d	i	0101	1	0
Store	cached read, store	0	p	0	CA	B	u
S32RI	cached read, store	0	p	0	CA	0	0
S32NB	cached read, store	0	p	0	CA	0	0
S32C1I	single atomic read/write	0	p	0	CA	0	0
Castout	cached write	0	d	0	1111	1	0

Notes:

- CA and B correspond to the values described in previous Request Attributes as a function of Cache Attribute tables.
- p corresponds to the P-bit being associated with a specific instruction, d it is not associated with a specific instruction.
- u corresponds to the user Request Attribute bits possible being set by an instruction.
- i corresponds to the auto-prefetch being due to hardware instruction prefetch.
- I-Fetch P-bit is not guaranteed to be accurate for fetches ahead of instructions that change privilege level. However inaccuracy is unlikely, as privilege level normally changes via exception/interrupt or an exception/return instruction in which case the correct value will be used.

User Defined Request Attribute Bits

Even though the bottom four bits of the Request Attribute are associated with user-defined load and store operations, it does not assure that the attributes will make it to the PIF when these load or store operations are executed.The specific cases where the Request Attribute set by a user defined load or store instruction is visible on the PIF are the following:

- A user defined store instruction that bypasses the cache or goes to a write through region will set the Request Attribute of the write request on the PIF.
- A user defined load instruction that bypasses the data cache will set the Request Attribute of the read request on the PIF.
- A user defined load or store instruction that is done to a writeback region and causes a cache fill will set the Request Attribute on the PIF read of the new cache line. Note that any castout that results from this cache line fill will have a Request Attribute of 0.

Examples of the usefulness of the user-controlled attribute field are:

- Have more user-level control over store operations that must receive a response from their final destination versus stores that can receive responses from an intermediate point, which can improve performance. This could be done if the built in "bufferable" attribute information is insufficient for a specific application, and more user level control is desired.
- Associate a priority with the request to be used by interconnects and memory controllers in allocating routing priority and bandwidth to requests.

These two examples are illustrated in the context of the Xtensa processor being a PIF master in the next section.

Xtensa TIE Instruction For Request Attribute Setting

The Xtensa processor can be extended with user-defined instructions using the Tensilica Instruction Extension (TIE) Language, and in particular a mechanism is provided to set the POReqAttribute output of outgoing PIF requests. The following examples assume familiarity with TIE, and with extending the processor.Refer to the *Tensilica Instruction Extension (TIE) Language User's Guide* and the *Tensilica Instruction Extension (TIE) Language Reference Manual* for more details.

The Xtensa processor can be extended to implement the user-defined attribute bits. Reads and writes from core instructions, as well as all castouts, drive 4'h0 onto the user-defined attribute bits. User-defined load and store instructions can drive an optional TIE interface, which will be directly issued onto the user-defined attribute bits of the PIF (if

the request is targeted for the PIF). The TIE interfaces are VAddr, MemData(In|Out)<n>,LoadByteDisable, StoreByteDisable, and PIFAttribute. Following are the interfaces' properties:

- Name: PIFAttribute
- Stage: M-stage
- Direction: "out" interface
- Width: 4 bits
- FLIX: Interface is available per load/store unit interface
- Restrictions: can be assigned from the regfiles, state, or immediates.
- Requirements: requires PIF 4.0 or PIF 3.2 Request Attribute configuration option

Example: Request Priority

The Xtensa processor issues all PIF requests with a POReqPriorty[1:0] level of 2'b10, which indicates a medium-high priority. A system may interconnect processors by connecting the outbound PIF ports of some processors to the inbound PIF ports of other processors through a PIF interconnection network, and it may be desirable to control the routing priority of the requests in the network. In this system the user may allocate the PIF request attributes as follows:

- POReqAttribute[1:0]: encodes a request priority to be used instead of the processor's ReqPriority[1:0] field.

Next, define the following store instruction, which takes priority as an argument:

```
immediate_range immpri 0 3 1
operation S32PRIORITY {in AR data, in AR addr, in immpri priority}
                {out VAddr, out MemDataOut32, out PIFAttribute}
{
     assign VAddr = addr;
     assign MemDataOut32 = data;
     assign PIFAttribute = {2'b00, priority};
}
```

The software can use this store operation to vary the priority of a store operation; the system designer must use the request attribute value to make appropriate routing and bandwidth allocation decisions.

13.3 Outbound-PIF Implementation Details

This section summarizes the outbound-PIF implementations.

13.3.1 Cache Misses, Refills, and Castouts

Cache misses, refills, and castouts are serviced through the processor's PIF. Consequently, the PIF is required (the No-PIF option must not be selected) in Xtensa processors configured with caches. The request type issued for a cache miss or castout depends on the cache-line size and the PIF data bus width, both of which are configurable. The number of transfers generated by an instruction-cache miss is computed as:

```
Transfers for I-cache miss = inst cache line size (in bytes) / PIF width
(in bytes)
```

The number of transfers generated by a data-cache miss or castout is computed as:

```
Transfers for D-cache miss = data cache line size (in bytes) / PIF width
(in bytes)
```

If the number of transfers is 1, then a single data request is issued to service the cache miss or castout. If the number of transfers is 2, 4, 8, or 16, then a block request is issued for a cache miss or castout and the block size is encoded on `ReqCntl`.

If critical-word-first is not configured, cache miss requests are issued with starting addresses that are aligned to the entire block size. If critical-word-first is configured, then cache miss requests are issued with starting address aligned to the critical PIF width of the request. Castouts will always be issued with starting addresses that are aligned to the entire block size.

13.3.2 No-Allocate and Bypass Requests

All load and store instructions that do not map to a local memory or the data cache and that have valid attributes are issued on the PIF. A load request to memory with the No Allocate access mode will check the data cache, and if the load target is not present in the data cache, it will make a request on the PIF - it does not allocate a cache line. A load request to memory with Bypass access mode does not check the instruction or data cache at all and immediately makes a request on the PIF. If a data load or store request is narrower than or the same size as the PIF, then typically a single PIF read or write request will be issued on the PIF and the byte enables depend on the size of the request. For example, one byte lane will be enabled for a byte-load transaction request.

If a data load request is wider than the PIF width, then a block request will be issued on the PIF. Stores that are wider than the PIF get broken up into several single writes. Note that when user-designed TIE store operations use byte enables that are not defined by the *Xtensa PIF Protocol Reference Manual*, and the PIF does not have arbitrary byte enables configured, even stores that are only as wide as the PIF may get broken up into several single write requests that each have allowed byte patterns (see the *Xtensa PIF Protocol Reference Manual*, *Tensilica Instruction Extension (TIE) Language Reference*

Manual, and *Tensilica Instruction Extension (TIE) Language User's Guide*). **Note**: Interrupts are not taken during accesses to PIF that are marked as "Cached no Allocate" or "Bypass". Note that this behavior may change in future releases.

Instruction fetches that do not access any local instruction memory or instruction cache are issued on the PIF as read requests. Depending on the relationship of the instruction fetch width to the PIF width these requests can either be single or block-requests.

13.3.3 Read-Conditional-Write Requests

Read-conditional-write requests are generated by executing `S32C1I` instructions. During the first transfer of the resulting transaction request, the request data bus will be driven to the value contained in the `SCOMPARE1` special register. During the second transfer of the request, data from the register specified by the `S32C1I` instruction is driven onto the request data bus. The data read from the target memory location during the read phase of PIF RCW operation is always returned. All read-conditional-write requests from the processor are 4-byte single data requests.

13.3.4 Unaligned Loads Handled by Hardware

Unaligned loads that are handled by hardware can produce two load requests on the PIF, if they span two data-memory widths. The resulting two loads are not guaranteed to be atomic, but will be performed in address order—in fact, the second load will not be issued until the first load has completed.

13.3.5 Multiple Outstanding Requests

Xtensa processors generate multiple outstanding requests. The Xtensa LX7 processor can have one instruction fetches and one load transaction request per load/store unit simultaneously outstanding for FLIX instructions. If prefetch is configured, up to 16 prefetch requests can be outstanding in addition.The maximum number of read transactions outstanding from an Xtensa LX7 processor with cache and prefetch is 19 (1 instruction fetch + 2 load + 16 prefetch). The maximum number of read transactions outstanding from an Xtensa LX7 processor with two load units and no pefetch is three (1 instruction fetch + 2 load). In addition, if write responses are configured and no prefetch, the processor allows 16 writes to be outstanding. If write responses are not configured, there can be no outstanding PIF write transactions because writes are considered to be completed once the write is issued on the PIF.

13.3.6 Xtensa Processor PIF Transaction Request Priority Usage

Xtensa processors issue PIF transaction requests with a medium-high priority (level `0x2`). Xtensa processors ignore the incoming priority bits of PIF responses.

13.3.7 Write Buffer

The write buffer queues the processor's PIF-directed store requests. This feature helps maximize execution bandwidth by decoupling the processor execution pipeline from PIF arbitration bubbles. The ordering requirements, number of entries are configurable parameters. When the PIF is unavailable (`PIReqRdy` is not asserted), store requests are queued in the write buffer.

A write buffer operation can be summarized by the following set of rules:

- The write buffer is a FIFO queue.
- Queued writes are issued to the PIF as soon as the PIF is available.
- Writes are prioritized relative to reads to either ensure that data reads and writes are issued over the PIF in program order or in a more relaxed mode that allows reads to bypass prior write requests to reduce read latency (see Table 13–59 on page 213).
- The write buffer will not combine write requests issued at different times. That is, the write buffer does not 'gather' separate write operations to the same address. Write requests to the address of a previously queued write transaction already in the write buffer will not affect the existing, queued writes.
- When the write buffer and store buffer are full, a new store request stalls the processor pipeline.

Write-buffer performance is a function of the write buffer depth, the mix of load and store requests, and the performance of the system interconnect.

13.3.8 Prefetch Write Buffer

For configurations that have a write back cache and prefetch to the L1 data cache, there is a prefetch write buffer that is used to contain the castouts that sometimes need to be done in order to bring a prefetched cache line into the L1 cache. This prefetch buffer can be 1 or 2 cache lines big. Once a prefetch castout line has been committed it arbitrates with the main write buffer to do writes to the PIF, while still maintaining correct memory ordering.

13.3.9 Request Prioritization and Ordering

Requests for the outbound PIF can be categorized into the following groups:

- Instructions fetches
- Data reads, including read-conditional-write requests
- Stores
- Castouts

These requests all arbitrate for a single outbound PIF port, and are prioritized in one of two different schemes based on the setting of the "Prioritize Load before Store" option. When the "Prioritize Load before Store" option is **not** selected, the following order is used to prioritize requests for the PIF (1 is the highest priority).

1. Stores
2. Data reads
3. Castouts
4. Instruction fetches

When the "Prioritize Load before Store" option is selected, the following order is used to prioritize requests for the PIF

1. Data reads, whose addresses do not overlap with any stores that are pending in the write buffer. See description following this list for details on how the *overlap* is determined.
2. Instruction fetches
3. Stores
4. Data reads that may overlap with stores pending in the write buffer.
5. Castouts

The priority of data reads in this second ordering depends on whether the data read *may* overlap with a store in the write buffer. In particular, eight address bits are compared with every pending store and if there is a match in those address bits, the load may overlap with an earlier store and should not be ordered before those stores. Comparing only eight bits of address may cause some false matches, which simply means that some reordering might not occur even though legal as described by the above definitions. The least-significant address bit that is compared is the address bit above the cache line size as described in the Table 13–59.

Table 13–59. Address Bits Compared to Prioritized Loads Before Stores

Cache Line Size	Data-Memory/Cache Port Width	Address Bits Compared
16 bytes	Any	[11:4]
32 bytes	Any	[12:5]
64 bytes	Any	[13:6]
128 bytes[1]	Any	[14:7]
256 bytes[1]	Any	[15:8]
No cache	4 bytes	[9:2]
No cache	8 bytes	[10:3]

Notes:

1. Controlled Feature (contact Cadence for details)

13.3.10 Response Ready

Xtensa LX7 processors, unlike previous versions of the Xtensa processors, will at times drive PORespRdy low in order to flow control responses. This is generally only for a short time as it waits for buffer space to become available, and is usually related to cache line refill logic that gets backed up. If early restart is not configured, then PORespRdy will always be high.

13.3.11 Maintaining Outbound Requests

Xtensa LX7 processors that drive a request using POReqValid, POReqAddr, POReqCntl, etc., but see a PIReqRdy low in response, will maintain this same request on subsequent cycles until PIReqRdy goes high and the request is accepted. Note that this is an Xtensa LX7 specific implementation feature, as the PIF protocol itself does allow the request to change or be withdrawn in the next cycle if PIReqRdy is low.

13.3.12 Bus Errors

A bus error can be reported for either an instruction or a data reference. Furthermore, an error can occur for either a cacheable or noncacheable reference. In the case of a cached reference, bad data may still be placed in the cache, but the cache line is marked as invalid. This scheme prevents the program from mistakenly using this data later on.

The PIF protocol does not support the early termination of block requests. Unless an address error is signaled, a response must have as many transfers as were requested and the entire response must be issued regardless of errors that may be signalled on individual transfers within the response.

Note that a given instruction may cause multiple exceptions. A bus-error exception might not be taken if other exceptions take priority. The following sub sections provide information about the relative priorities of instruction exceptions.

Instruction Reference Bus Error

An address- or data-bus error returned on an instruction reference has the following effect:

1. The instruction that caused the error is tagged with an indication that it caused a bus error. If the reference is to a cacheable location, the returned data may be placed in the instruction cache, even with the error. The cache line is marked invalid.
2. If a bus error exception is taken, the exception PC (EPC1) will point to the offending instruction and the exception virtual address register (EXCVADDR) will point to the memory address location of the offending instruction.

Note: When the instruction straddles a word boundary, EPC1 and EXCVADDR may point to different words. If the instruction is re-executed and the bus error occurs again, the exception will be taken again.

3. The exception cause register (EXCCAUSE) will be set to 12 if a data error was reported on the PIF or to 14 if an address error was reported on the PIF. If both an address and data error are reported on the PIF, then the cause register is set to 14 to signify an address error.
4. It is possible for the processor to get a bus error on a cache line that is never executed. Such an error will not, however, generate a bus error exception.

Load Bus Error

An address- or data-bus error returned on a load-cache miss or uncached/bypass cache load has the following effect:

1. The load that caused the error is tagged with an indication that it generated a bus error. If the reference is to a cacheable location, the returned data may be placed in the data cache, even with the error, but the cache line is marked invalid.
2. If a bus-error exception is taken, the EPC1 register will point to the offending load instruction and the exception virtual address register (EXCVADDR) will point to the load address that caused the error. If the instruction was killed, no bus-error exception is generated. If the instruction is re-executed and the bus error occurs again, then the exception will be taken again.
3. The exception cause register (EXCCAUSE) will be set to 13 if a data error was reported on the PIF or to 15 if an address error was reported on the PIF. If both an address and data error are reported on the PIF, then the cause register is set to 15 to signify an address error.

Read-Conditional-Write Bus Error

If a bus-error exception is taken on a response to a read-conditional-write PIF request, the EPC1 register will point to the offending instruction, and the exception cause register (EXCCAUSE) will be set to 13 if a data error was reported on the PIF or to 15 if an address error was reported on the PIF. If the instruction was killed, no bus-error exception is generated. If the instruction is re-executed, and the bus error occurs again, then the exception will be taken again.

Note: It is impossible to distinguish between an error on the read-and-compare transaction and an error on the conditional-write transaction. Therefore, the entire operation is assumed to have failed.

Bus Error on Write Request

Because store requests commit in the processor's pipeline before the corresponding write request is issued on the PIF, it is not possible to take a precise exception, meaning an exception that is tied directly to the instruction that caused it, for bus errors generated by write requests. Two methods of reporting bus errors for write requests are supported. Both involve interrupts:

1. PIF slaves configured to received and return write responses can encode a bus error in a write response. The processor can then be configured to allocate a bit of the interrupt register to "Write Bus Error" signaling. The processor will raise this interrupt bit for each write response that signals the occurrence of a bus error. This interrupt is subject to the same masking and prioritization as any other configured interrupt, but it is an internal interrupt and does not have a separate interrupt input port. It can only be signalled by a write response that encodes a bus error.
2. If no interrupt is allocated for write responses, then write transactions are assumed complete when an external agent acknowledges the processor's write request with the `PIReqRdy` signal. The system designer may reserve an external interrupt pin to signal bus errors generated by write transactions. The external agent can assert this external interrupt, and the interrupt handler then decides how to handle the bus error.

13.4 PIF Master Port for iDMA

When iDMA is configured, a dedicated PIF master port is instantiated for it. iDMA utilizes PIF protocol 4.0 features to achieve data movement efficiency. It requires or uses the following properties:

- iDMA requires its PIF port to support arbitrary byte enables.
- iDMA use a single read ID and a single write ID to enforce the ordering of requests and responses.
- By programming the descriptor control word, the programmer can select iDMA POReqPriority.
- By programming the iDMA control register, the programmer can select the maximum allowed PIF block size to be used by iDMA.
- iDMA can automatically decompose unaligned data stream into multiple aligned PIF requests.
- iDMA can be programmed to issue up to 64 outstanding PIF requests.
- iDMA reports bus errors via the status register and the iDMAError interrupt.

14. Xtensa Configurations with No PIF Bus Interface

Xtensa processors configured without a PIF (No PIF processors) bus interface can be used for SOC designs requiring minimum gate count. However, No PIF processor configurations do not support caches because Xtensa processors perform cache-line load, write-through, and write-back operations via the PIF. No PIF processors can be configured with two different memory management units only, those being Region Protection and Region Protection with Translation.

References to physical addresses outside the local memory range made by a processor configured without a PIF results in either an instruction-fetch error or a load/store error, depending on the attempted transaction.

Without a PIF to which the diagnostic infrastructure can attach arbitrarily sized memories, No PIF processors may have very limited memory availability, depending on the size of local memories configured. The level of software and diagnostic support is constrained accordingly. Available XTOS and C library features may be limited, according to what can fit in memory. It may be necessary to use specific LSPs to minimize memory overhead. To understand what objects get pulled together to create the software image, refer to the *Xtensa Linker Support Packages (LSPs) Reference Manual* sections about Object Files and Libraries; in particular, review issues related to using the MIN-RT and TINY LSPs.

Further, to save on both Instruction memory and data memory, No PIF processors can be configured with only 16 AR registers, that is, without AR register windowing. In this case, only the call0 ABI is used. This eliminates the set of window overflow/underflow handlers in instruction memory, and minimizes call stack overhead by only allocating stack space for registers that are used rather than allocating fixed size call frames. This is counteracted by the call0 ABI's need for extra code at entry and exit of each function, but this extra cost is usually easily outweighed by the savings in the case of small programs that can fit in only a few kB of instruction memory.

Following are the memory requirements for the smallest No PIF processors with No Windowing:

- 4K instruction memory
- 1K writable data memory (Data Ram or XLMI)
- Maximum of two higher interrupt levels (interrupt, debug, or NMI)
- 16 AR Registers (call0 ABI only)

The memory requirements for No PIF processors with Windowing and up to seven interrupt levels are:

- 8K instruction memory

- 1K writable data memory (Data Ram or XLMI)

All Xtensa software will run on No PIF processor configurations. However, test-software restrictions are application-code dependent; some applications may not fit in local memory. Minimal C programs using the default XTOS runtime package and the default sim LSP may require as much as 8-Kbytes of instruction RAM and 8- Kbytes of data RAM. Running C programs in a smaller system may require use of the TINY LSP or a custom LSP.

15. Xtensa Cache Interface Bus

Caches use fast, small memories to buffer the processor from the slower and larger memories generally located external to the SOC (system on a chip). The processor core's caches store the data and instructions that a program is immediately using, while the majority of other data resides in slower main memory (RAM or ROM). In general, the instruction and data caches can be accessed simultaneously, which maximizes processor bandwidth and efficiency.

Note: The cache ports described in this chapter may be connected only to blocks with memory semantics. The processor will occasionally issue speculative loads to these ports and actual loads in the instruction stream are occasionally generated multiple times. There are no signals to indicate whether the loads are speculative or real. Stores to the ports described in this chapter are never speculative but may be delayed by several cycles due to write buffers and queues within the processor.

15.1 Instruction and Data Cache Description

The Xtensa processor accesses information stored in the cache when there is a *cache hit* in the cache look-up sequence. A hit occurs when there is a match between the cache's physical address tags and the physical address. When there is a cache hit, the processor fetches the required instruction or data from the associated instruction or data cache and the operation continues.

A *cache miss* occurs when the cache's physical address tags do not match the physical address of the access. In the case of a cache miss, the desired instruction or data is not present in the cache and a cache-line worth of instructions or data is retrieved from the slower external memory to load the missing information into the cache. When the cache miss is caused by a read or an allocate-store operation, the processor executes a cache-line refill sequence involving a pipeline replay and the generation of a request for new data from external memory. The requested information must be retrieved from the external memory before processor operation can continue. The amount of data retrieved on a miss depends on the configured cache-line size and ranges from 16 to 256 bytes.

Note: Refer to Appendix A.4 for an explanation about replay.

The Xtensa processor's cache interface is separate from, but intimately related to, the PIF. The first load access of a given instruction or data word will cause a cache miss and will therefore be read from main memory via the PIF. However, after the initial access is completed, a copy of the retrieved information is kept in the faster cache memory. Sub-

sequent accesses to this location are now returned from the cache, thereby speeding system access and producing better performance. Software written as a series of tight instruction loops benefits from this instruction and data locality.

Because loads from the data cache compete with stores to it, stores pass through a store buffer in the processor so that loads and stores can somewhat overlap. The store buffer is a shared queue of write operations to the data-cache, data-RAM, and the XLMI ports. Stores can complete when the cache is not otherwise occupied with a another higher priority cache operation.

To keep instructions and data flowing into the execution units in an optimal manner, the Xtensa processor core offers various cache options such as direct-mapped vs. set- associative caches and a range of cache and cache-line sizes.

Each cache actually consists of two memory arrays. One memory array stores the cached information and is referred to as the *data array*. The other memory array, referred to as the *tag array*, keeps track of the physical addresses of the information held in the data array. Thus, for an Xtensa processor with direct-mapped instruction and data-caches, there are four cache memory arrays: the instruction-cache data array, the instruction-cache tag array, the data-cache data array, and the data-cache tag array.

15.1.1 Data Cache Banking

When more than one load/store unit is configured with caches, the cache data arrays are separated into either two or four banks based on the lower bits of the address.

For example, a 32-bit wide 8k cache banked four ways would consist of four 2k memories. And bits[3:2] of the address would determine which bank is activated when the data cache is accessed.

While the cache data arrays are banked in a multi-load store unit configuration, the tag arrays are duplicated on a per load store unit basis.

As a result, each load store unit can simultaneously access the data cache as long as the accesses are to different banks. If there are enough banks configured, the chance that any given load/store unit will contend with another is drastically reduced which ensures full speed operation.

15.1.2 Memory-Error Protection

The Xtensa processor’s cache interface can optionally be configured with memory error protection using either parity or an error-correcting code (ECC). The parity scheme provides single bit error detection and the ECC scheme provides single-bit error detection and correction and 2-bit error detection.

If memory-error protection is configured, the interface ports to each cache RAMs' data-array and tag-array (for both read and write data) are supplemented with memory-check bits that are read back from the cache with the cached data and are written to the cache memory in the same way as the regular data. The parity-protection scheme for the instruction cache uses one additional memory bit per 32 bits of data-array width and one additional memory bit for each tag array.

There are two different options for parity-protection on the data cache depending on the level of protection required. One can either choose to have one extra memory bit per byte of data array width or one extra memory bit per 32 bits of data array width. Protecting the tag arrays only requires a single extra memory bit.

For the ECC protection scheme, the instruction cache uses seven extra memory bits per 32 bits of data array width and seven extra bits for each tag array. Like the parity option, there are two different options for ECC protection on the data cache.

The ECC protection scheme for the data cache can either use five extra memory bits per byte of data array width or seven extra memory bits per 32 bits of data array width. Protecting the tag arrays requires seven extra memory bits.

15.1.3 Instruction Cache Width Selection and Word Enables

Normally the instruction cache width should be the same as the instruction fetch width, either 32-, 64-, or 128-bits. The instruction fetch width is the amount of data fetched from the cache on each instruction fetch read access, and there is rarely any benefit to having the instruction cache width wider than this, since wider memories usually consume more power. One exception is when instruction cache refill time is critical, since a wider instruction cache reduces this refill time, assuming a wider PIF interface is also used.

If the instruction cache width is wider than 32-bits, the design outputs an instruction word enable for every 32-bit word, which can be used for the following purposes:

1. The cache test instruction SICW can write just a single 32-bit word even if the instruction cache width is 64-bits or 128-bits wide. If the word enables are not used then the 32-bit word is replicated and written across the full instruction cache width. Writing a single 32-bit word allows finer control over software based memory testing.
2. Instruction cache memory power can be reduced by only enabling the minimum width required for instruction fetch. Specifically:
    a. If the instruction fetch width is 32-bits, and the cache instruction width is 128-bits, then instead of using a single 128-bit wide memory array that uses `ICache<way>En` as the enable signal, the design can use four 32-bit wide memory arrays that each use one of the four `ICache<way>WordEn` bits as the enable signal.

b. If the instruction fetch width is 32-bits, and the cache instruction width is 64-bits, then instead of using a single 64-bit wide memory array that uses ICache<way>En as the enable signal, the design can use two 32-bit wide memory arrays that each use one of the two ICache<way>WordEn bits as the enable signal.

c. If the instruction fetch width is 64-bits, and the cache instruction width is 128-bits, then instead of using a single128-bit wide memory array that uses ICache<way>En as the enable signal, the design can use two 64-bit wide memory arrays, one that uses ICache<way>WordEn[0] as the enable signal, and one that uses ICache<way>WordEn[1] as the enable signal. Alternatively one could use four 32-bit wide arrays and use a word enable for each of them.

Note that currently the use of the instruction word enables is optional, but future OCD (on-chip debug) software may require that instruction word enables be used in order to implement faster download times.

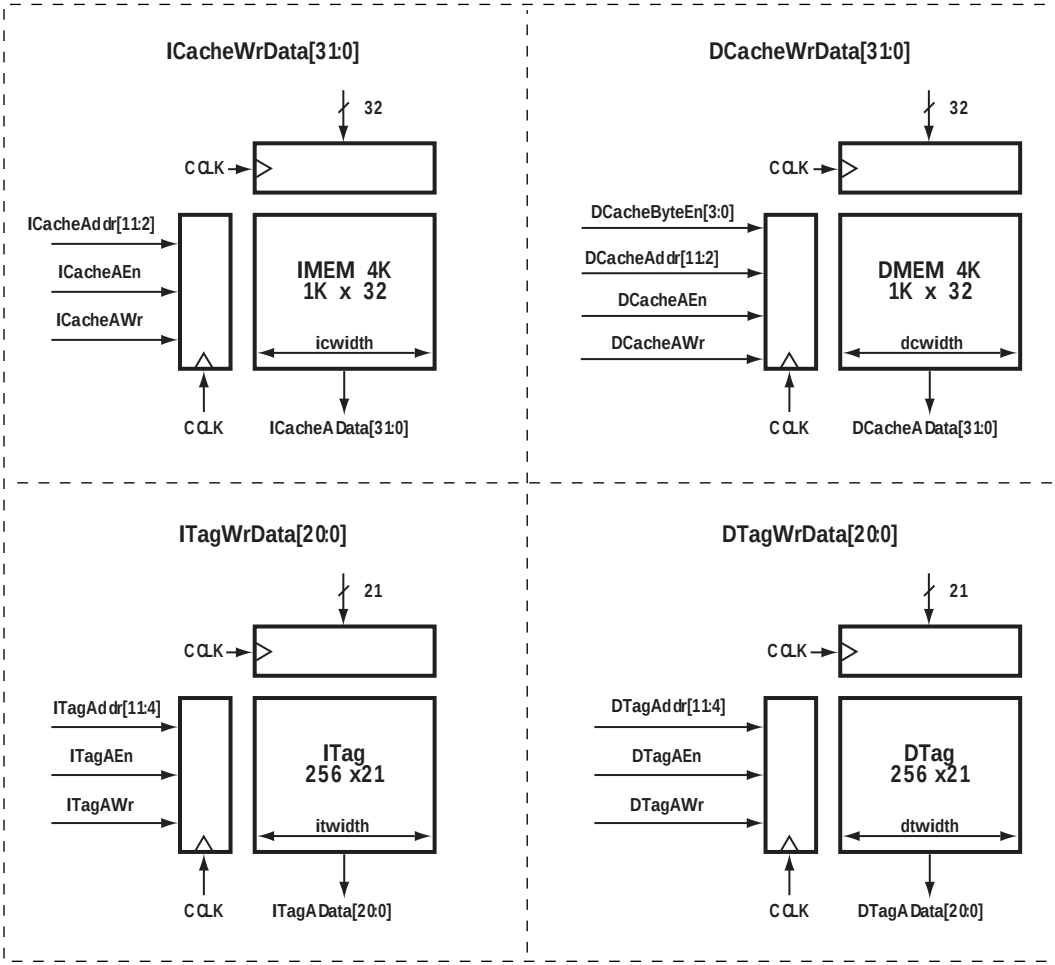


Figure 15–32. ICache, DCache, ITag, and DTag Configuration for Direct-Mapped Caches

Figure 15–32 shows the cache arrays for a system with 4-Kbyte instruction and 4-Kbyte data caches (16-byte line size). In this example, both cache data arrays are organized as 1K by 32 bits. Both cache tag arrays are organized as 256 x 21 bits.

Note: The width of the tag array depends on the data array size.

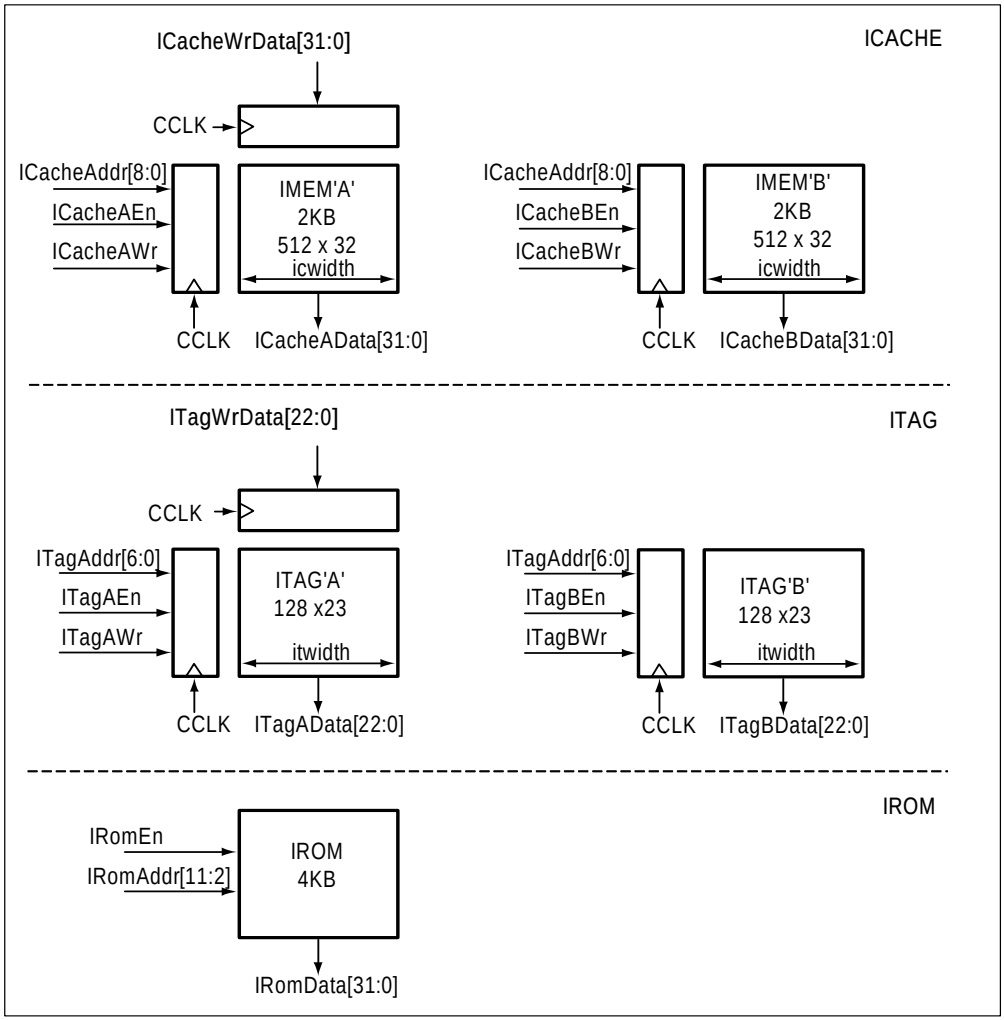
For a 4Kbyte memory array, the processor's 32-bit memory addressing is partitioned such that the lower 12-bits of the address index into the 4Kbyte data array and the remaining upper 20-bits of the address serve as the cache tag. In addition to the tag itself,

the cache tag array also stores a valid bit. In other configurations, the tag array may also store a line-replacement-algorithm bit (for set-associative caches), a lock bit (when the locking option is configured), and a dirty bit (for write-back caches) for each address tag.

In Figure 15–32, CCLK is a buffered version of CLK (external clock), synchronized to PCLK in the processor's core. Note that the Xtensa core supports synchronous SRAM (rising- edge clocked) for all local memories. Cadence recommends that the entire memory subsystem associated with an Xtensa processor design be synchronous to the PCLK domain. See Chapter 34, "Xtensa Processor Core Clock Distribution" for more details about PCLK.

The address, data, and control lines to the processor's RAM arrays are unidirectional and are always driven (they are never allowed to float). Xtensa processors require synchronous RAM arrays. That is, the address, enable, write, and data-input lines are presented to the RAM arrays with the appropriate set-up time with respect to the rising edge of a clock signal. In the case of a read, data from the cache must become available on the following clock cycle (for an Xtensa LX7 processor with a 5-stage pipeline) and two cycles later (for an Xtensa LX7 processor with a 7-stage pipeline). Writing data to the RAM array requires assertion of the data, address, enable, and write lines with the prescribed set-up time with respect to the rising edge of the clock. The actual write to the RAM array will occur during the next cycle for processors with either 5- or 7-stage pipelines.

Figure 15–33 shows an example of a two-way set-associative instruction cache. The two cache ways share an address bus and the write-data bus, but they have separate enable and write lines and read-data buses. Also shown in the figure is an instruction ROM that exists in parallel with the instruction cache.



The example above is for ICache = 4KB with 16B/Line; IRom = 4KB

Notes:
1. itwidth = 32 – log2 [2KB] + 1 (valid bit) + 1
2. CCLK is a buffered version of CLK (external clock), synchronized to PCLK in the Xtensa core

Figure 15–33. Example of 2-Way Set Associative Instruction Cache and Instruction ROM

Figure 15–34 shows an example of a 4KB 2-way set associative data cache with two banks and two tag arrays. Note that cache banking is only supported in configurations with more than one load/store unit. Each load/store unit has its own tag array and the

data arrays are banked based on the lower bit(s) of the address. The two cache ways share an address bus and a write-data bus on a per bank level, but they have separate enable, write lines, and read-data buses.

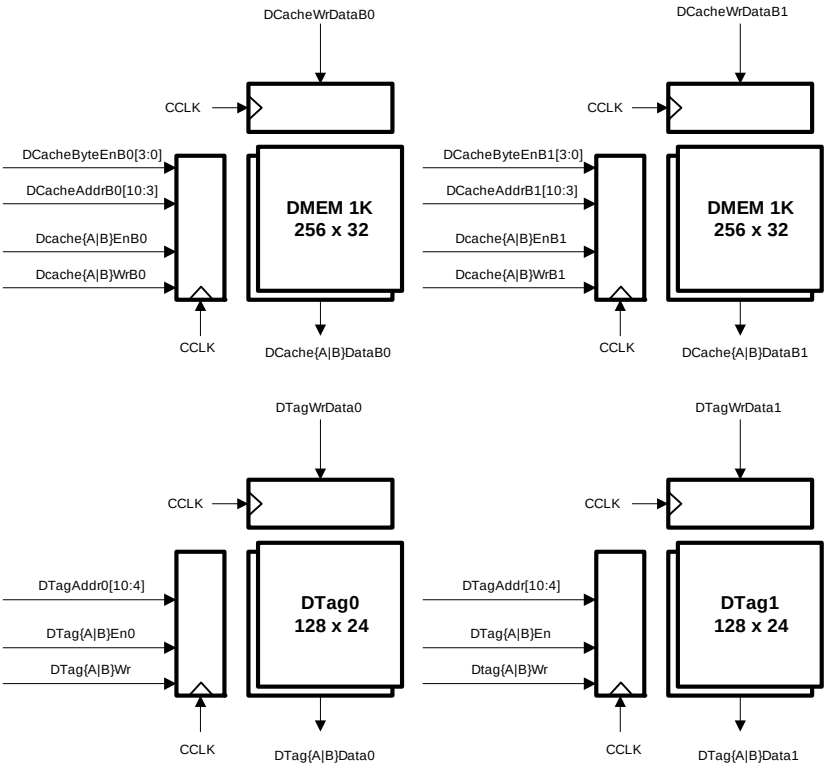


Figure 15–34. 4KB 2-Way Set Associative Data Cache with 2 Banks and 2 Tag Arrays

15.2 Cache Port Signal Descriptions

For lists of the instruction- and data-cache interface signals, see Table 10–17, “Instruction Cache Interface Port Signals,” on page 134 and Table 10–20, “Data Cache Interface Port Signals,” on page 138.

Note: See Section 35.6 on page 629 for information about cache interface AC timing.

16. Cache Access Transaction Behavior

This chapter describes Instruction and Data Cache behavior and timing. It also describes different cache operations, and includes representative timing diagrams.

16.1 General Instruction-Cache Timing

Figure 16–35 shows the relationship between the memory control and address, and the data coming from cache memory for the 5-stage-pipeline version of the Xtensa LX7 processor.

The address and control signals are presented during one cycle and the cache returns data during the next cycle. For a write cycle, the processor presents the write data during the same cycle that it asserts the address and control signals. The tag-array write that occurs during cycle 3 in Figure 16–35 invalidates the tag of way A in the cache. This might occur during execution of an IHI instruction, for instance.

The indices into the data and tag arrays (`ICacheAddr` and `ITagAddr`) depend on the size of the caches, their associativity, and the cache-line size, and are made up from bits taken from the virtual address (see Chapter 23, “Local-Memory Usage and Options” on page 425).

Caches are physically tagged, so the tag arrays contain data from the physical addresses of the cached instructions and data to detect a cache hit. Certain special instructions such as `III`, `IIU`, `SICT`, `LICT`, `SICW`, and `LICW` access the instruction-cache memory arrays as transaction targets, thus treating the caches as memories rather than caches.

Figure 16–35 shows the same memory transaction, but for the 7-stage-pipeline version of the Xtensa LX7 processor which expects one extra cycle of delay before data returns from the cache. Note that the 2-cycle cache-memory implementation must be such that a write followed immediately by a read to the same address must return the newly written data, not the old data.

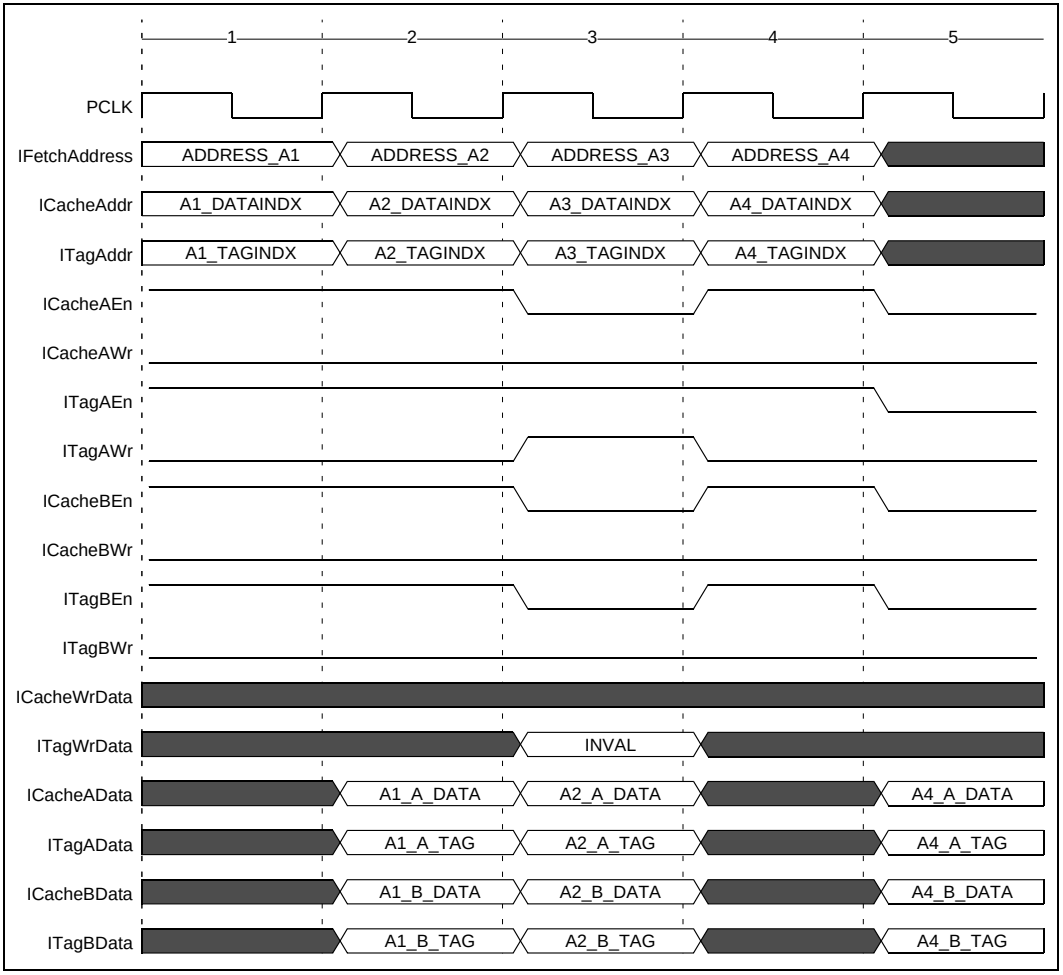


Figure 16–35. Cache Access Example (5-Stage Pipeline)

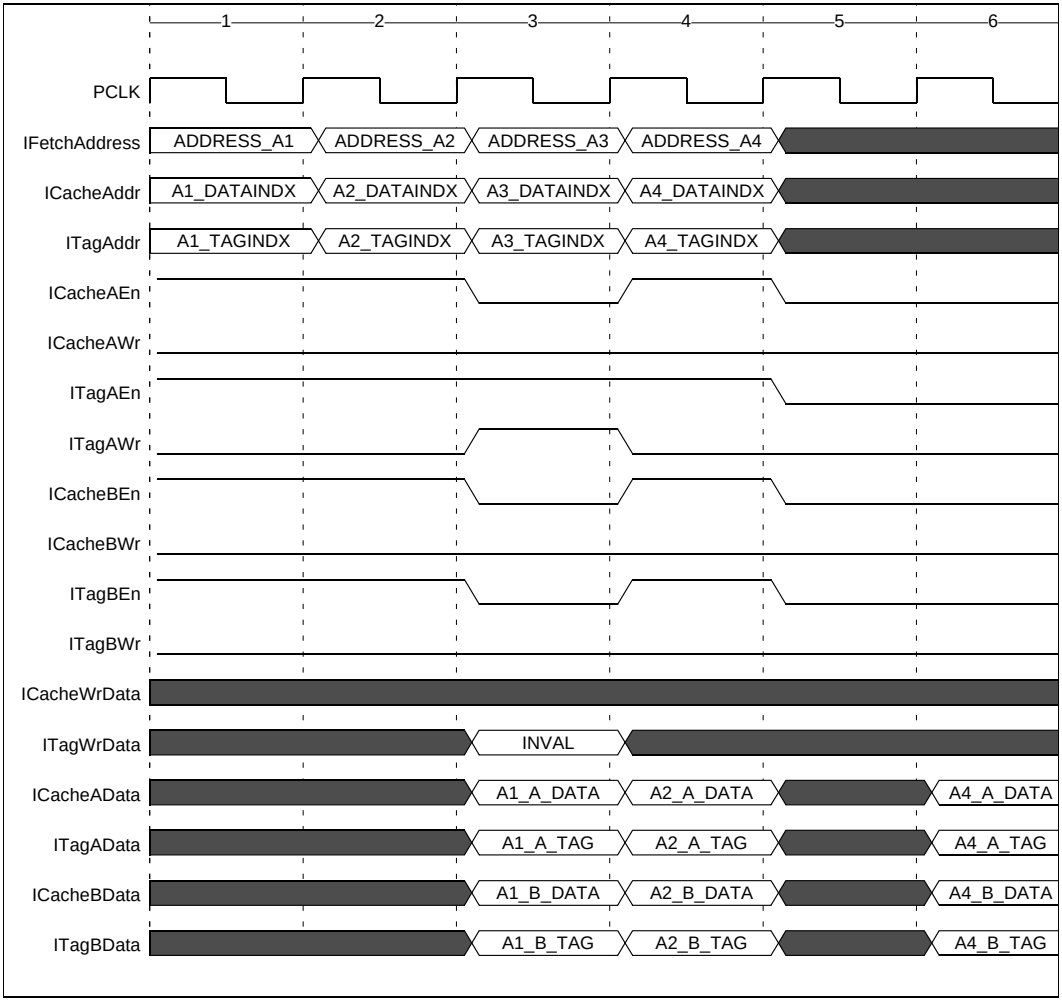


Figure 16–36. 7-Stage Pipeline Cache Access Example

16.2 Instruction-Cache Line Fill

Figure 16–37 shows a typical cache-line-fill sequence for a 2-way cache with no Early Restart. A cache miss is signalled only when the pipeline attempts to use bytes to form an instruction in the R-stage of the processor pipeline (refer to Appendix A “Xtensa Pipeline and Memory Performance”), which can occur several cycles after the invalid data is fetched from the cache. In this example, the processor has fetched data from address 64 that has a tag address of 19 in cycle 1, and the data is first used during cycle 3. A request is made to the external interface to fetch the cache line. The external interface in turn begins a PIF transaction (not shown), and the PIF returns the data for the speci-

fied cache line. The absolute minimum latency for the first word of the cache line to be written to the cache is three cycles, and this latency is shown in Figure 16–37. The actual memory latency depends on the activity on the external interface, the PIF latency, and the external memory latency.

The cache controller's LRF (least-recently filled) algorithm selected cache way A as the replacement target, so when the cache-line data arrives from external memory, it is stored in cache way A while cache way B remains idle. Four PIF-widths of data are required to fill the cache line in this example, and these correspond to the four words written to the cache data array. These four words are not necessarily returned on successive cycles, as shown in Figure 16–37 by the gap between WORD1 and WORD2. On the first word of the cache fill, the tag of the cache line is written to it's correct value assuming the cache line fill will be successful. If a bus error were to occur on the PIF response then the tag will be re-written to be invalid. After the fill completes, execution resumes with both cache ways being accessed. Now that the required information resides in the cache, a cache hit occurs when the processor replays the instruction and tries to use data from the target address.

Note that an instruction-cache line fill is not canceled before it completes. The cache fills are decoupled from the pipeline and write the caches opportunistically, and an exception or an interrupt will not affect a cache fill.

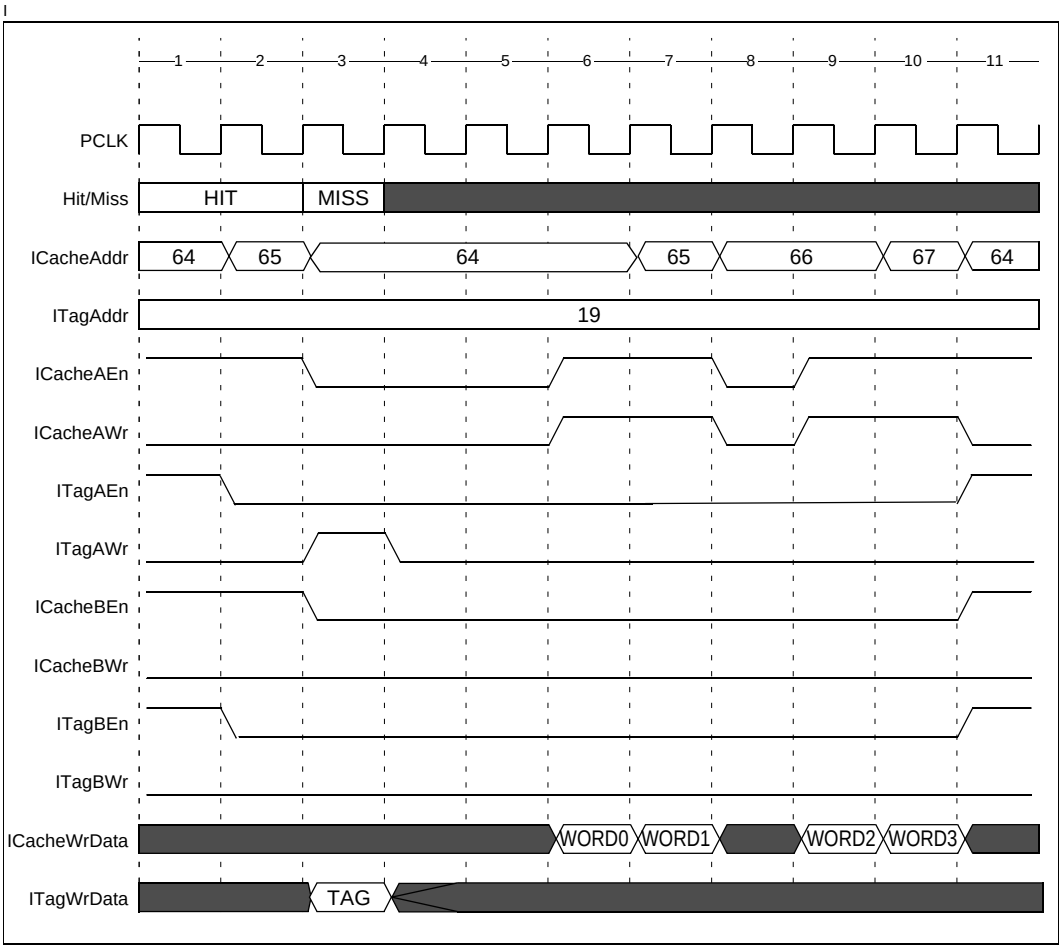


Figure 16–37. Instruction Cache-Line Miss with No Early Restart

The above example assumes no Early Restart, so that the fill completes before the pipeline resumes. In an Early Restart configuration, the pipeline can restart as soon as the critical data is returned, and the refill occurs in the background in cycles where the instruction cache interface is not used as shown in Figure 16–38. The pipeline restarts once the data is available from the Response Buffer, and the instruction cache does not need to be enabled. A background refill cycle is shown in cycle 9.

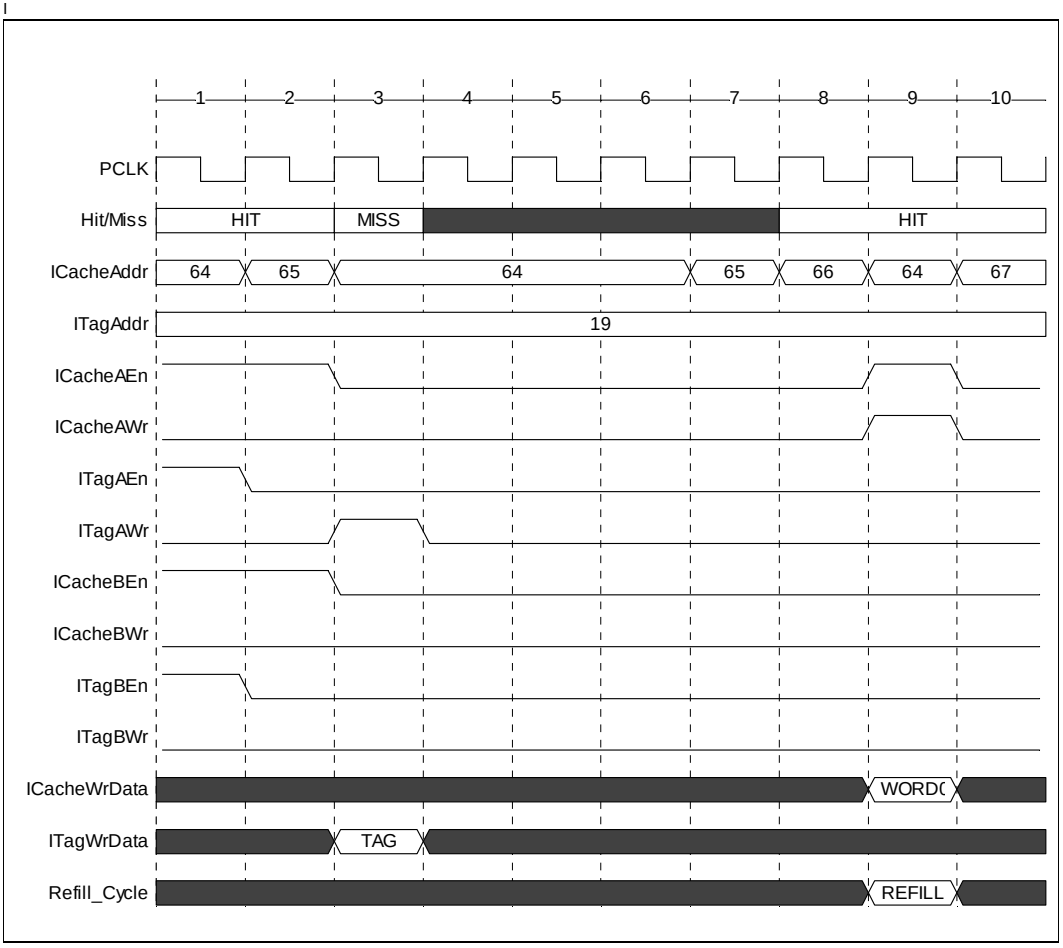


Figure 16–38. Instruction Cache-Line Miss with Early Restart

16.3 Instruction-Cache Access Instructions

In addition to reading from and writing to the instruction cache during instruction execution and instruction cache-line fills, the Xtensa processor's ISA provides a number of instructions that directly manipulate the instruction-cache arrays. These instructions include `IHI`, `III`, `IIU`, `IPFL`, `IHU`, `LICT`, `LICW`, `SICT`, and `SICW`. The `IHI` instruction checks to see if there is a cache hit for a specific address and it invalidates the cache tag if there is a cache hit. The `III` instruction invalidates one tag in each cache way for a specific cache index. For processor configurations with instruction-cache locking, the `IPFL` instruction fetches a specified cache line (if it is not already in the cache) and locks the line in the cache. The `IHU` instruction checks to see if there is a cache hit and

whether the cache line is locked. If it is locked, it unlocks that cache line without invalidating it. The IIU instruction unlocks and invalidates one cache line at a specified index and way.

Note: This behavior is different than that of the DIU instruction, which operates on the data cache and unlocks but does not invalidate the cache lines.

The cache-test instructions allow the tag arrays and data arrays to be read and written directly. The LICT and SICT instructions load from and store to the tag array respectively, and the LICW and SICW instructions respectively load from and store to the data array.

Note: LICT, SICT, LICW, and SICW instructions are meant for manufacturing tests only, and should not be run out of the instruction cache as they are testing the instruction cache.

The cache instructions are divided into three categories:

- Instructions that store to the arrays (IHI, III, IHU, IIU, SICT, SICW)
- Instructions that load from the arrays (LICT and LICW)
- The IPFL instruction, which can cause a cache-line fill in addition to writing the tag array

An example of a store to the tag array appears in Figure 16–39, which shows an IHI instruction followed by an add instruction. Only the address specifically related to the IHI instruction is shown, although the tag array is also being used during other cycles. During IHI instruction's E stage, the processor reads the tag array to see if there is an address match. Assuming a hit is detected, the processor writes to the tag during the W stage of the IHI instruction and the tag is invalidated. The tag invalidation causes the processor to flush its pipeline, so the add instruction following the IHI is replayed.

A store to any of the cache arrays caused by the IHI, III, IHU, IIU, SICT, or SICW instructions follows this same pattern where the write to the array during the instruction's W stage is followed by a replay of the next instruction. The writes caused by III, IIU, SICT, and SICW do not depend on a cache hit.

Note: The write also occurs during the W stage for Xtensa LX7 processors with a 7-stage pipeline.

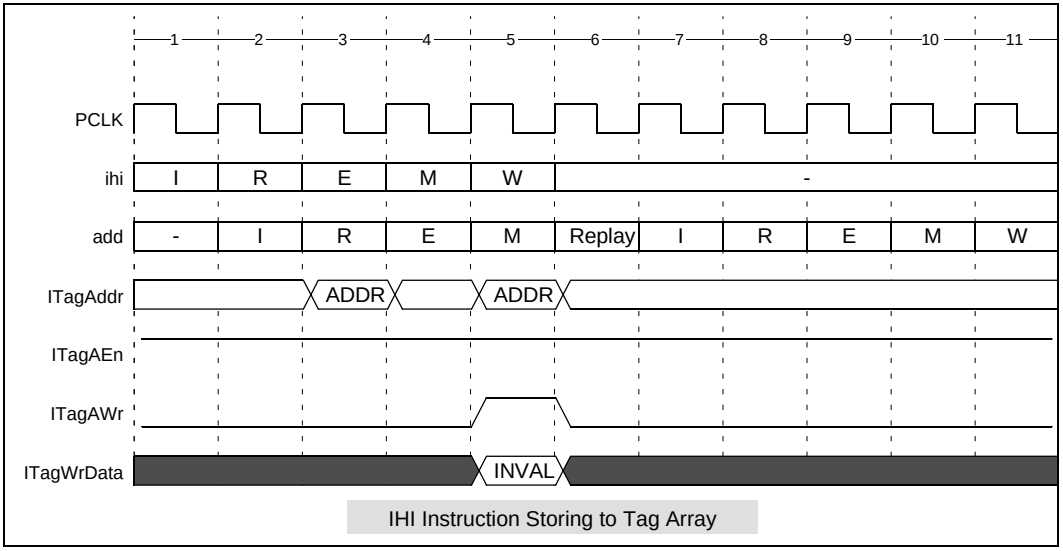


Figure 16–39. Store to Instruction Cache Array Example

Figure 16–40 shows an example of a read from the cache array by an `LICW` instruction. Only the address specifically related to the `LICW` instruction is shown, although on other cycles the data array is also being used. During the processor's M stage, the `LICW` instruction reads the cache, and the data coming from the array is captured in a temporary register one or two cycles later, depending on whether the processor has a 5- or 7-stage pipeline. During the W stage, the `LICW` instruction is replayed, and during the replay (refer to Appendix A.4 for replay explanation), the data is taken from the temporary register and loaded into the register file.

The `IPFL` instruction combines elements of the cache-line-fill transaction and the write-to-tag-array transaction. If the desired cache line is already in the cache, the `IPFL` instruction writes to the tag array to lock the cache line (assuming that there is more than one unlocked way at that cache index), behaving much like the IHI example. If the desired cache line is not in the cache, then the IPFL instruction first generates a cache-miss request. When the cache-line fill completes, the processor replays the `IPFL` instruction and writes to the tag array to set the lock bit when it reaches the W stage.

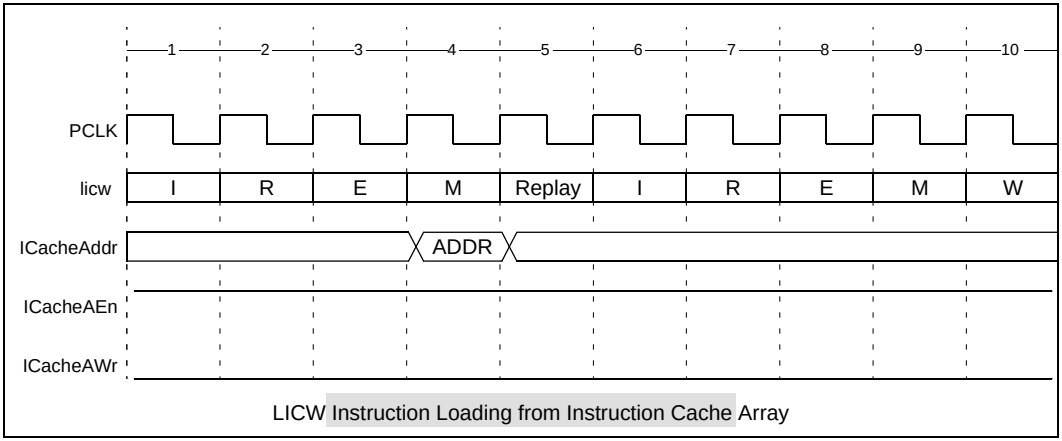


Figure 16–40. Load From Instruction Cache Array

16.4 Instruction-Cache Memory Integrity

The instruction cache can optionally be configured with memory parity or error-correcting code (ECC). Extra bits are read from and written to the cache data arrays and the cache tag arrays, and are used to detect data errors that may occur in the cache. The parity option detects single-bit errors. ECC detects and corrects single-bit errors and detects double-bit errors.

If a memory error is detected during an instruction fetch (correctable or uncorrectable), an exception is signalled when pipeline execution uses a byte from that instruction fetch. Effectively, the instExc bit of the MESR register is tied to 1.

If a memory error occurs when fetching an instruction from the instruction cache (see Section 16.3), the hardware takes the following special actions for the following cases:

- `IHI`, `III` (when instruction cache locking is configured), `IHU`, and `IPFL`: All these instructions read the instruction-cache tags and will take an exception in the pipeline's W-stage when a memory error is detected on any instruction-cache tag array at the specified tag index.
- `LICT` and `LICW`: These instructions read the tag or data array directly. An uncorrectable error on the read, or a correctable error on the read when the `DataExc` bit of the `MESR` register is set, will cause an exception in the pipeline's W-stage of the replay of these instructions. If a correctable error is detected and the `DataExc` bit of the `MESR` register is not set, these instructions will perform two replays rather than the usual one to log the memory error and will also correct the error so that these instructions will retire correctly without taking an exception on the second replay.

Note: Although ECC can correct the data before it's used by the processor, the erroneous data still resides in the target cache location. It will be corrected if the processor reads this location again but to correct the error in memory, the processor must explicitly write the corrected value back to memory.

16.5 General Data Cache Timing

The following cache interface and processor interface timing diagrams illustrate write-through and write-back data-cache transactions for both early restart/critical word first and non-early restart configurations. Note that these diagrams apply to both direct-mapped and set-associative cache configurations.

Table 16–60 outlines the key points that occur during a cache hit or a cache miss, for either a load or a store, and for either a write-through or a write-back cache configuration. Write-back cache configurations can have either an allocate or no-allocate policy for store misses. This table applies to both direct-mapped and set-associative caches.

Table 16–60. Data Cache Write-Through and Write-Back Load/Store Transactions

	Hits		Misses	
	Write-Through	Write-Back	Write-Through	Write-Back
Load	■ Data loaded from cache	■ Data loaded from cache	■ Line allocated to cache ■ Data loaded from cache	■ Line allocated to cache ■ Victim line is cast out to memory if dirty ■ Data loaded from cache
Store	■ Data written to cache ■ Data written to external memory	■ Data written to cache	■ Data written to external memory	■ Victim line is cast out to memory if dirty ■ If allocate on store miss: ■Line allocated to cache ■Data written to cache ■ If no-allocate on store miss: ■ Data written to external memory

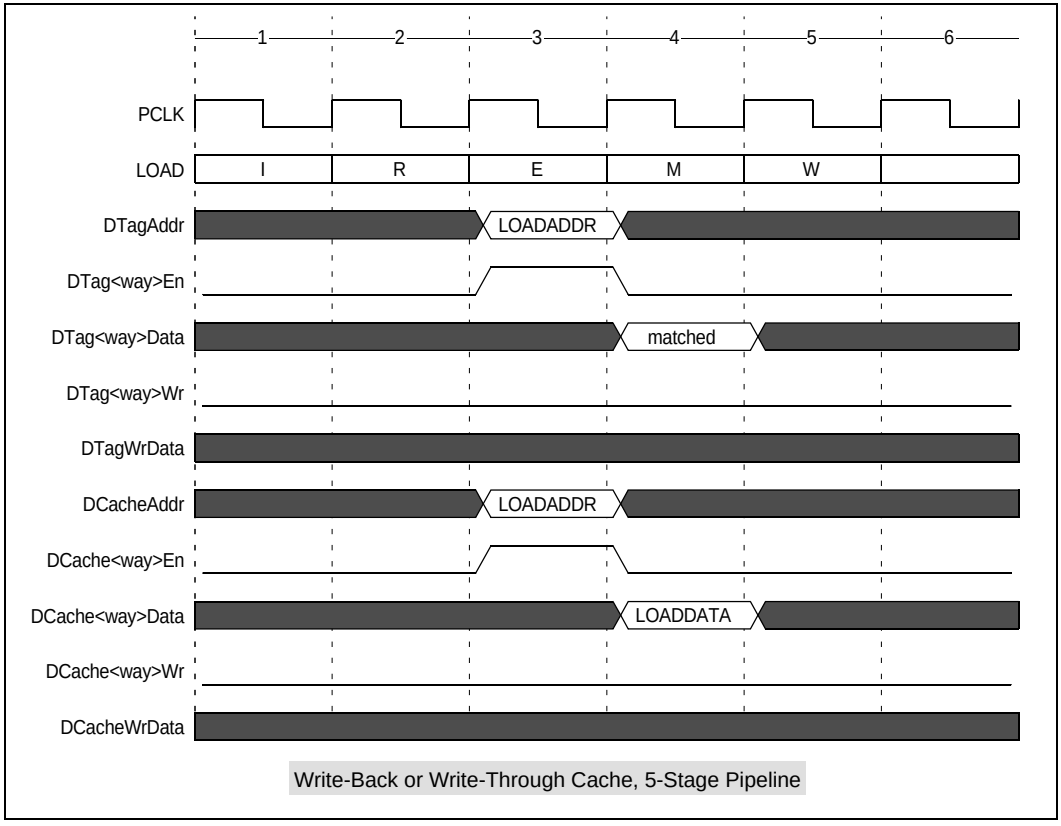


Figure 16–41. Cache-Load Hit (5-Stage Pipeline)

Figure 16–41 shows a cache-load hit for a Xtensa LX7 processor with a 5-stage pipeline.

During the load's E stage (cycle 3), the processor asserts `DTagAddr`, `DTag<way>En`, and `DCache<way>En`. It reads `DTag<`*`way`*`>Data` and `DCache<`*`way`*`>Data` during the load's M stage (cycle 4). The following figure shows a load hit because there is an address match with one of the cache tags during cycle 4. Tag data in only one of the cache ways should match. Loaded data or data read from data cache is on `DCache<way>Data` during cycle 4.

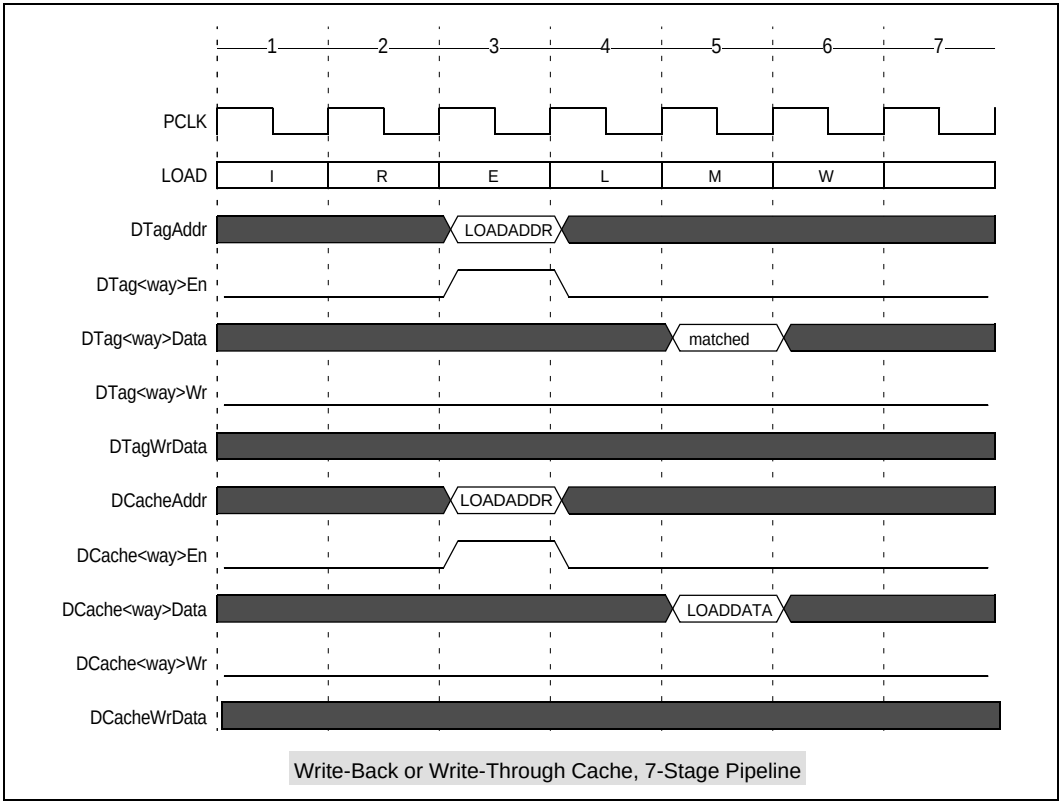


Figure 16–42. Cache-Load Hit (7-Stage Pipeline)

Figure 16–42 shows a cache-load hit for a processor with a 7-stage pipeline. During the load's E stage (cycle 3), the processor asserts `DTagAddr`, `DTag<way>En`, and `DCache<way>En`. It reads `DTag<`*`way`*`>Data` and `DCache<way>Data` during the load's M stage (cycle 5). Because this pipeline has seven stages, the memories have a 2-cycle read latency, which causes the data to arrive during cycle 5. Figure 16–42 shows a load hit because there's an address match with one of the cache tags during cycle 5. Tag data in only one of the cache ways should match.

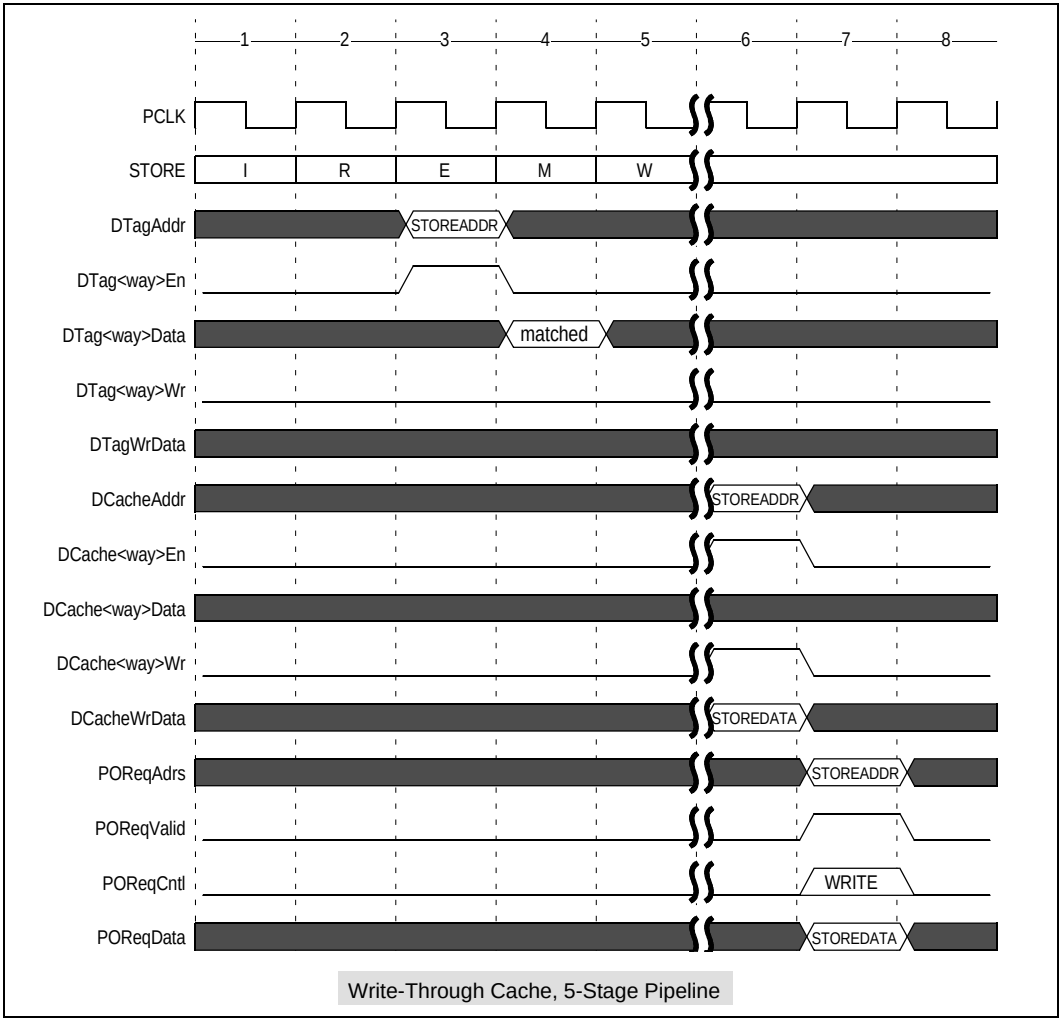


Figure 16–43. Cache-Store Hit (5-Stage Pipeline)

Figure 16–43 shows a cache-store hit for processors with a 5-stage pipeline.

During the store's E stage (cycle 3), the processor asserts `DTagAddr`, and `DTag<`*`way`*`>En`, and reads `DTag<`*`way`*`>Data` during the store's M stage (cycle 4). Figure 16–43 shows a store hit because there is a match with `DTag<`*`way`*`>Data` and the store's address. Tag data in only one of the cache ways should match. Also note that the store shown in cycle 6 could occur as early as cycle 5, but it may be delayed some number of cycles depending on what else is happening in the pipeline. Subsequently, the processor asserts `POReqValid` to start writing `POReqData` over the PIF at `POReqAdrs`.

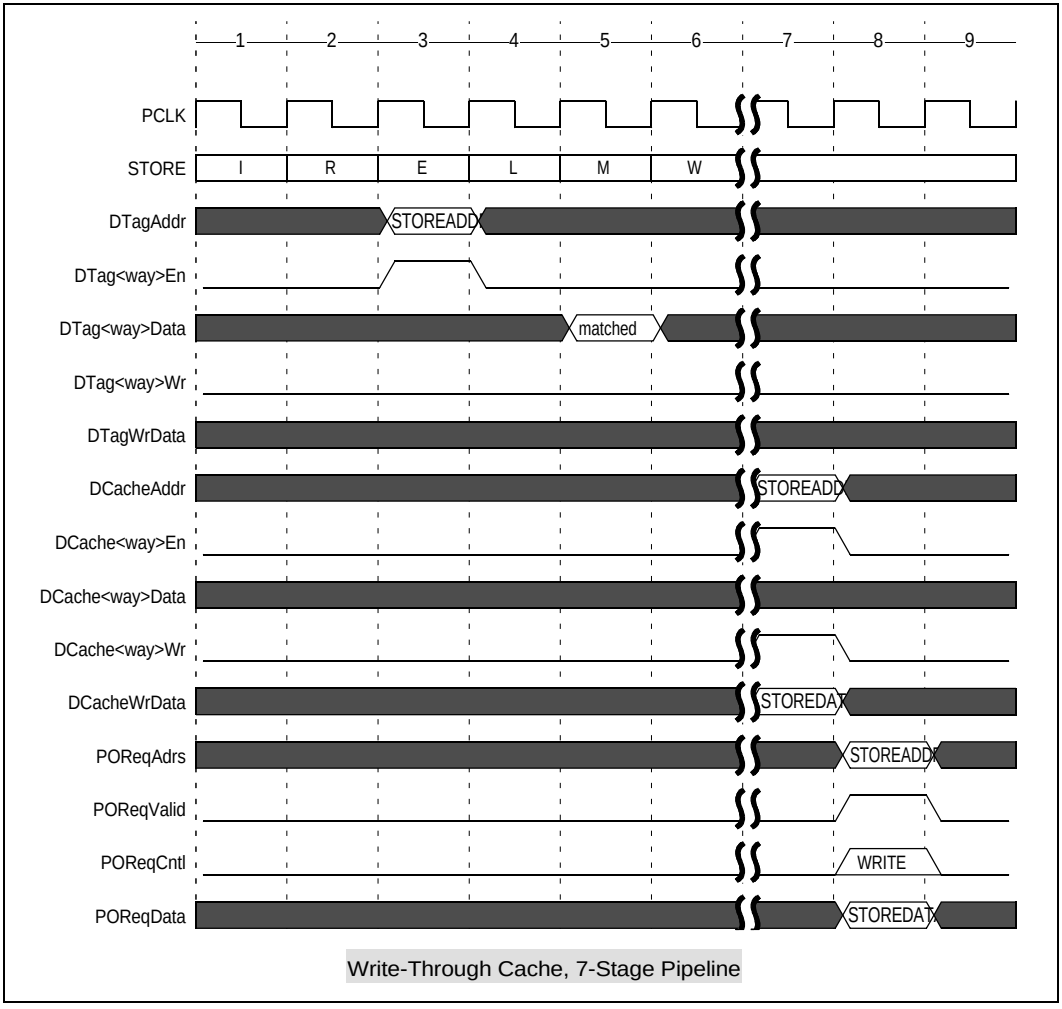


Figure 16–44. Cache-Store Hit (7-Stage Pipeline)

Figure 16–44 shows a cache-store hit for a processor with a 7-stage pipeline. During the store's E stage (cycle 3), the processor asserts `DTagAddr`, and `DTag<`*`way`*`>En` and reads `DTag<`*`way`*`>Data` during the store's M stage (cycle 5). Figure 16–44 shows a store hit because there is a match with `DTag<`*`way`*`>Data` and the store's address. Tag data in only one of the cache ways should match. Also note that the store could occur as early as cycle 7 but may be delayed some number of cycles depending on what else is happening in the pipeline. Subsequently, the processor asserts `POReqValid` to start writing `POReqData` over the PIF at `POReqAddrs`.

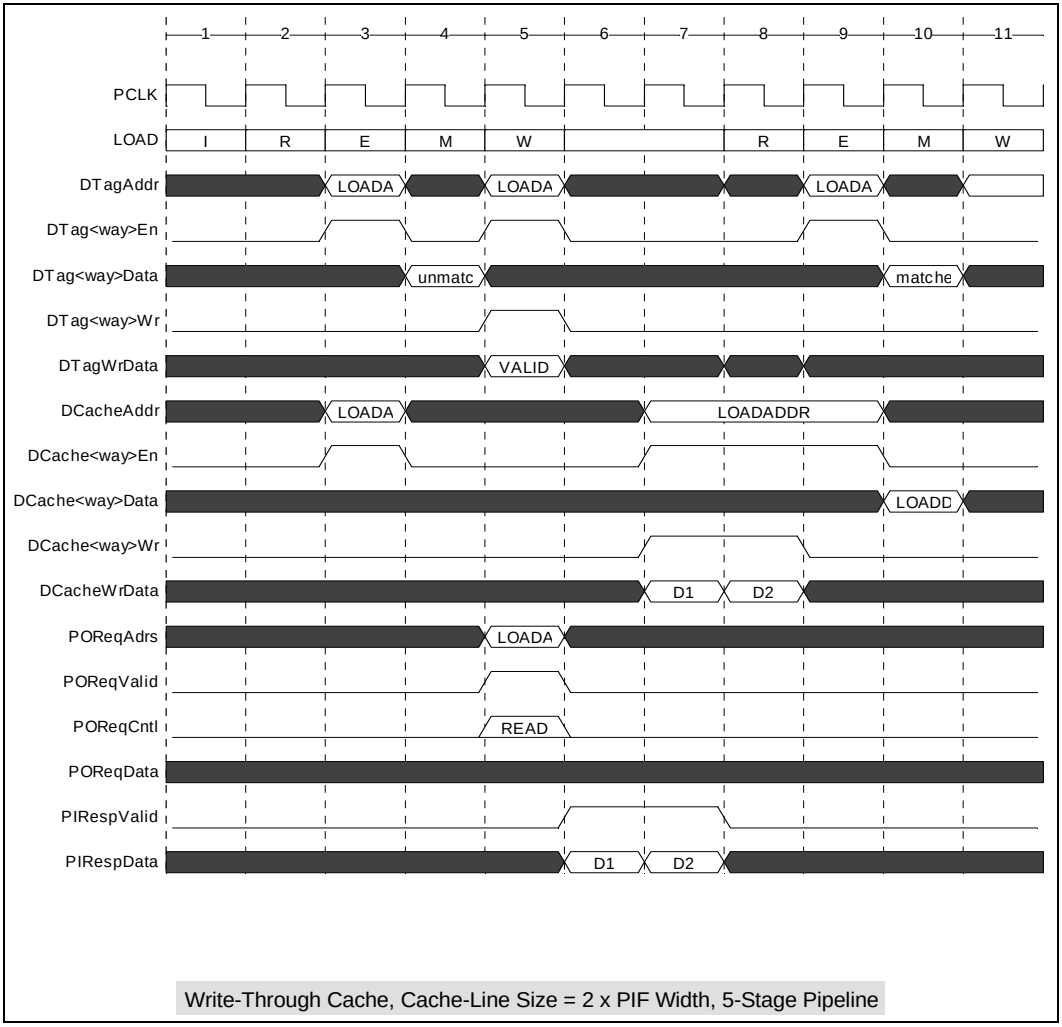


Figure 16–45. Cache-Load Miss (5-Stage Pipeline No Early Restart)

Figure 16–45 shows a cache-load miss for a processor with a 5-stage pipeline.

During the load's E stage (cycle 3), the processor asserts `DTagAddr`, `DCache<way>En`, and `DTag<way>En`, and reads `DTag<way>Data` during the load's M stage (cycle 4). Figure 16–45 shows a load miss because there is a mismatch with `DTag<way>Data` and the load's address. This line miss needs to be allocated to cache and generates a PIF request to memory. Note that data return on the PIF in cycle 6, followed by cycle 7, is the optimal case in this transaction.

During cycle 5, the processor asserts POReqValid, POReqAdrs, and POReqCntl to initiate a PIF read from external memory to satisfy the cache refill. Depending on the PIF latency, cycles could be added between cycles 5, 6, and 7. Note that Figure 16–45 shows the instruction being held in the R stage of the execution pipeline, but it could also be held in the P stage of the fetch pipeline if the load instruction cannot be re-fetched immediately.

Note: See Appendix A for more information on the Xtensa processor's pipeline stages.

PIRespData, DCache<way>Data, DCache<way>En, DCache<way>Wr, and DCacheWrData show activities for a line size that is twice the PIF width coming back to the cache. Cycle 10 shows DTag<way>Data matching the load address and data being loaded via DCache<way>Data. Note that the cache line is written into only one way of the cache.

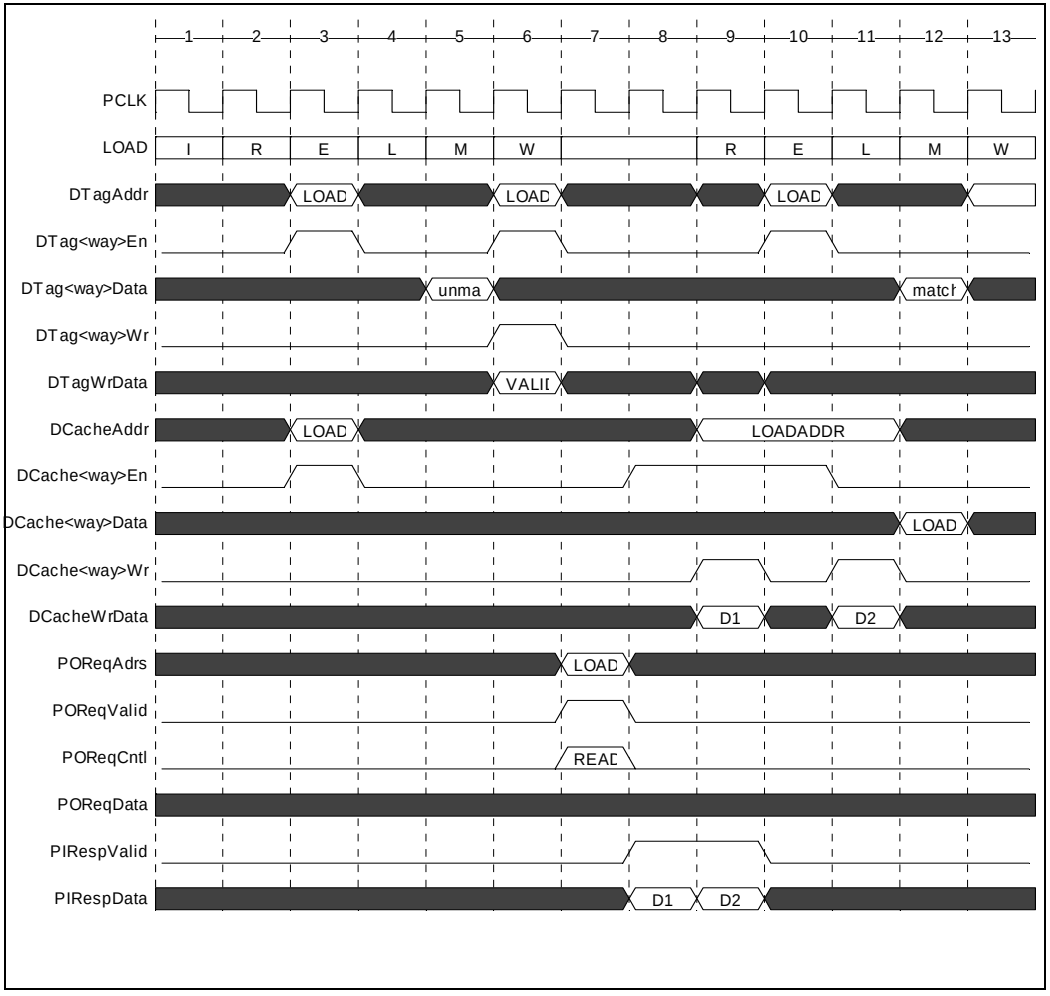


Figure 16–46. Cache-Load Miss (7-Stage Pipeline No Early Restart)

Figure 16–46 shows a cache-load miss for a processor with a 7-stage pipeline. During the load's E stage (cycle 3), the processor asserts `DTagAddr`, `DCache<way>En`, and `DTag<way>En`, and reads `DTag<way>Data` during the load's M stage (cycle 5). Figure 16–46 shows a load miss because there is a mis-match between `DTag<way>Data` and the load's address. This cache-line miss needs to be allocated to cache and generates a PIF request to memory. Note that data return on the PIF in cycle 7, followed by cycle 8, is the optimal case in this transaction. During cycle 6, the processor asserts `POReqValid`, `POReqAdrs`, and `POReqCntl` to initiate a PIF read from external memory to satisfy the refill. Depending on the PIF latency, cycles will be added between cycles 6, 7, and 8. Note that Figure 16–46 shows the instruction being held in the R stage of the execution pipeline, but it could also be held in the P stage of the fetch

pipeline if the load instruction cannot be re-fetched immediately. PIRespData, DCache<way>Data, DCache<way>En, DCache<way>Wr, and DCacheWrData show activities of a line size that is twice the PIF width coming back to be written to cache. Cycle 12 shows DTag<way>Data matching the load address and data being loaded via DCache<way>Data. Note that the cache line is written to only one way of the cache.

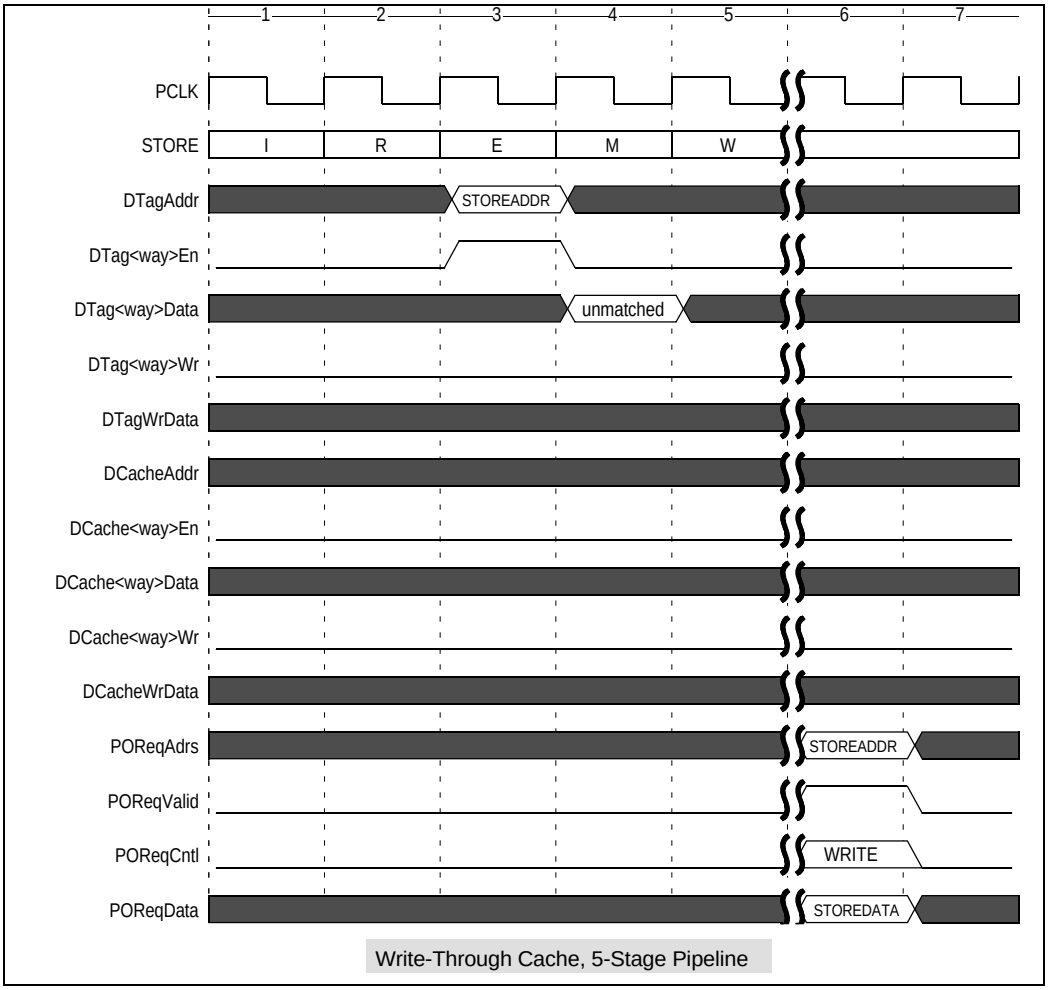


Figure 16–47. Write-Through Cache-Store Miss (5-Stage Pipeline)

The figure above shows a cache-store miss for a processor with a 5-stage pipeline and write-through cache. During the store's E stage (cycle 3), the processor asserts DTagAddr and DTag<*way*>En and reads DTag<*way*>Data during the store's M stage (cycle 4). This figure shows a store miss because there is a mismatch between DTag<way>Data and the load's address. Because the cache line is not in cache and it resides in a write-through cache region, the line is not allocated to cache and is instead

written directly to external memory. This figure represents the optimal case for this transaction. The processor asserts POReqValid, POReqAdrs, and POReqData to start the write to the PIF at cycle 6 or later depending on PIF traffic.

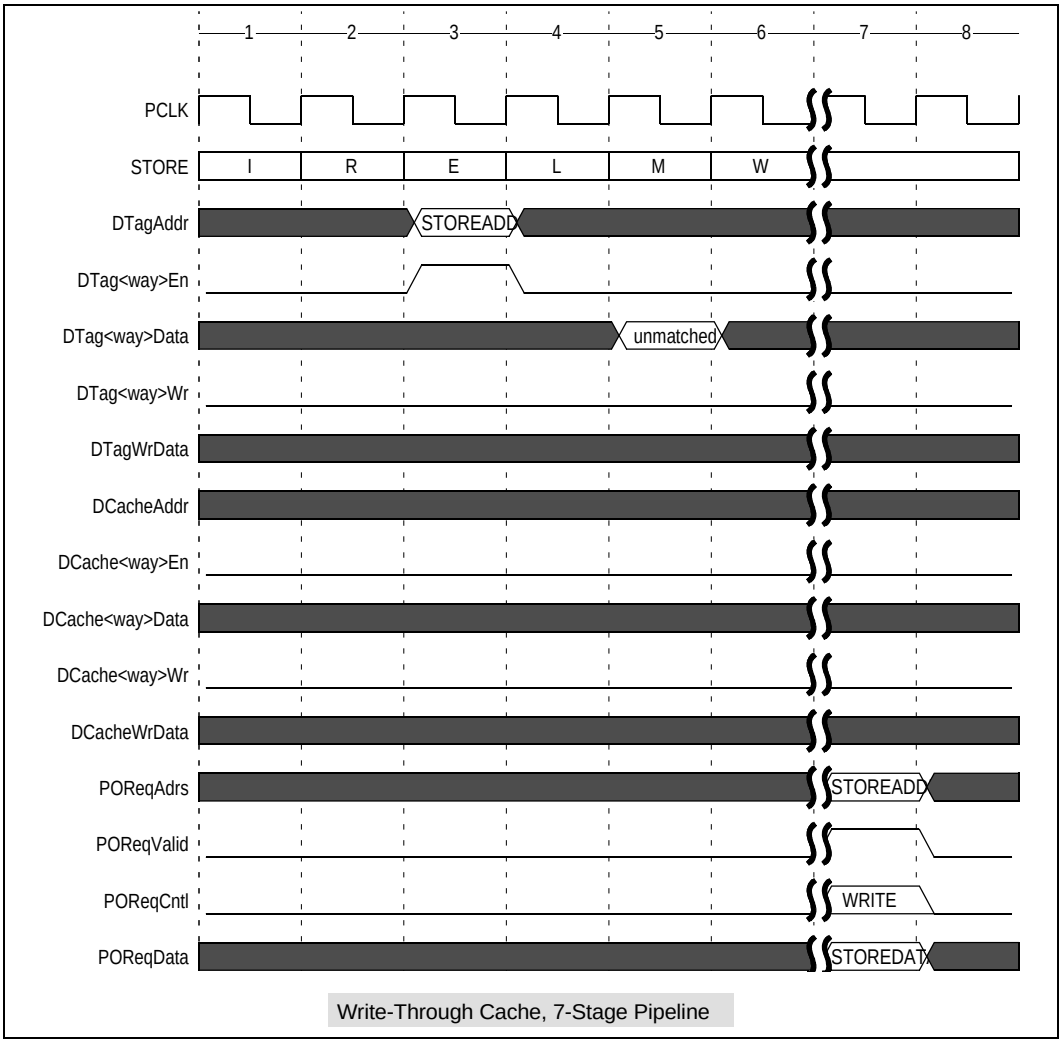


Figure 16–48. Write-Through Cache-Store Miss (7-Stage Pipeline)

Figure 16–48 shows a cache-store miss for a processor with a 7-stage pipeline and write-through cache. During the store's E stage (cycle 3), the processor asserts DTagAddr and DTag<*way*>En, and reads DTag<*way*>Data during the store's M stage (cycle 5). Figure 16–48 shows a store miss because there is an address mismatch with the cache tag. Because the line is not in cache and resides in a write-through cache region, the line is not allocated to cache and is instead written directly to external memory.

Figure 16–48 represents the optimal case for this transaction. The processor may assert POReqValid, POReqAdrs, and POReqData to start the write to the PIF at cycle 7 or later, depending on PIF traffic.

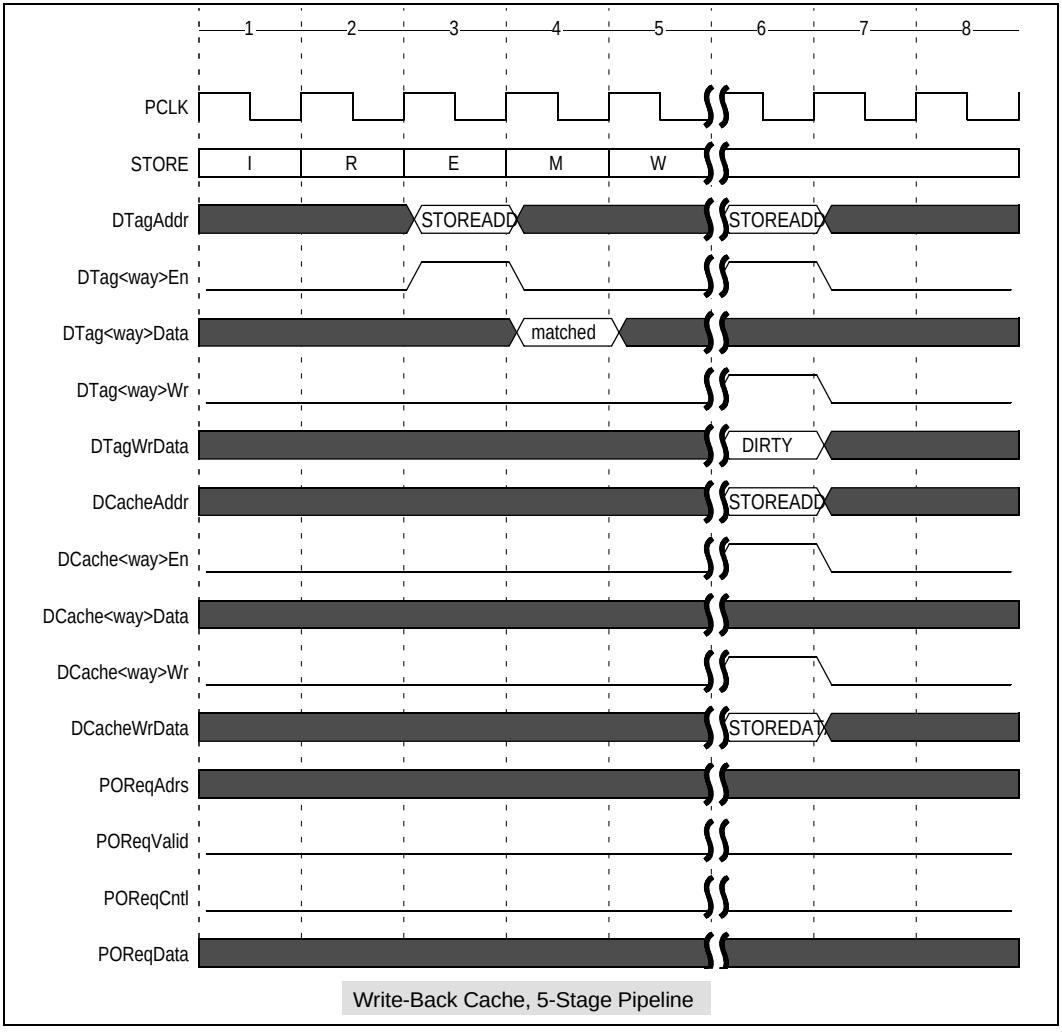


Figure 16–49. Cache-Store Hit (5-Stage Pipeline)

The figure above shows a cache-store hit for a processor with a 5-stage pipeline and write-back cache. During the store's E stage (cycle 3), the processor asserts DTagAddr and DTag<way>En and reads DTag<*way*>Data during the store's M stage (cycle 4). This figure shows a cache-store hit because there is a match between the address and the cache tag. Tag data in only one of the cache ways should match. Because the store modifies the cache line, the dirty bit is set via DTagWrData during cycle 6. The processor drives DCacheAddr and DCacheWrData and asserts DCache<*way*>En and

DCache<*way*>Wr to store the data. Although this figure illustrates that tag and data are being updated simultaneously, these updates may not necessarily happen in the same cycle. The tag and data updates depend on the states of the tag buffer and store buffer, respectively.

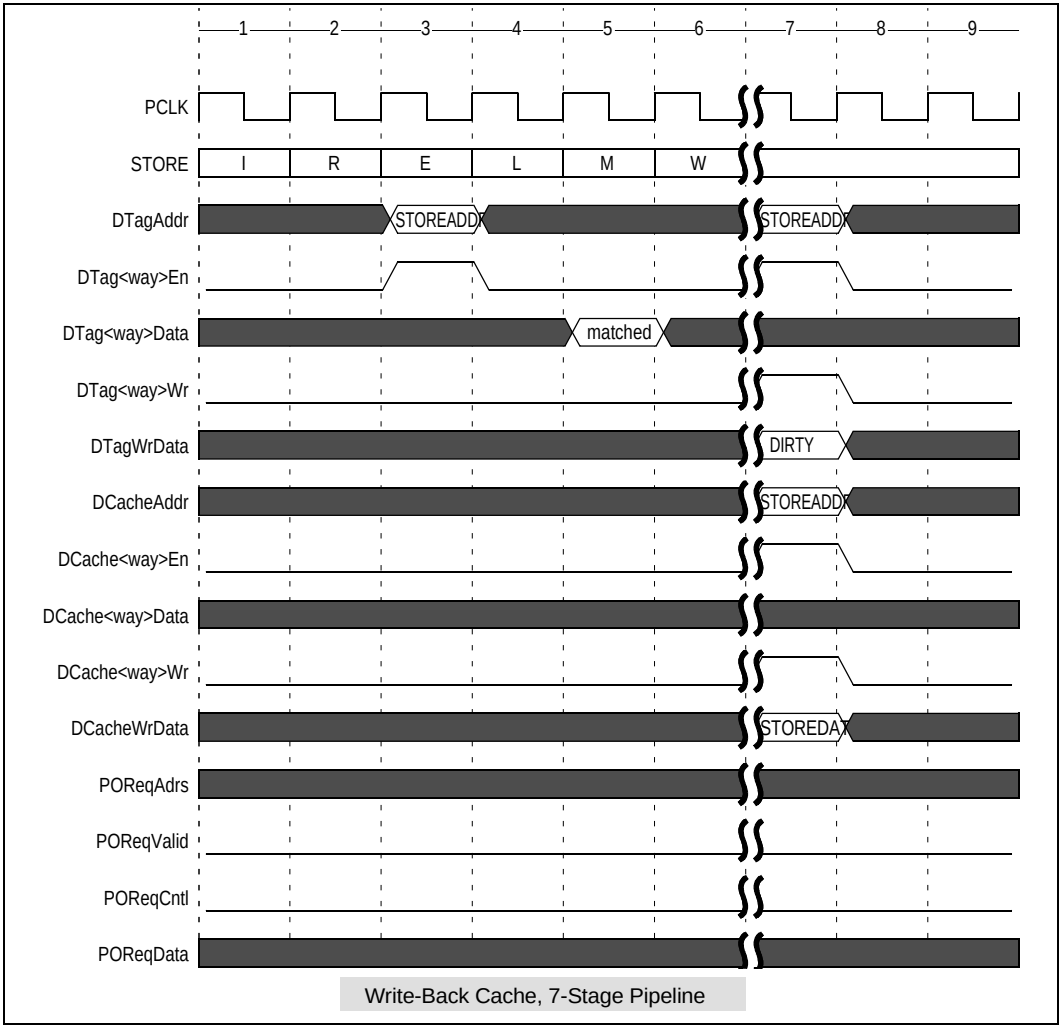


Figure 16–50. Cache-Store Hit (7-Stage Pipeline)

Figure 16–50 shows a cache-store hit for a processor with a 7-stage pipeline and write-back cache. During the store's E stage (cycle 3), the processor asserts DTagAddr, and DTag<way>En and reads DTag<*way*>Data during the store's M stage (cycle 5). Figure 16–50 shows a store hit because there is a match between the address and the cache tag. Tag data in only one of the cache ways should match. Because the cache line has been modified, the dirty bit is set via DTagWrData during cycle 7. The processor

drives DCacheAddr and DCacheWrData, and asserts DCache<*way*>En and DCache<*way*>Wr to store the data. Although Figure 16–50 shows that tag and data are updated simultaneously, these updates may not necessarily happen in the same cycle. The tag and data updates depend on the states of the tag buffer and store buffer, respectively.

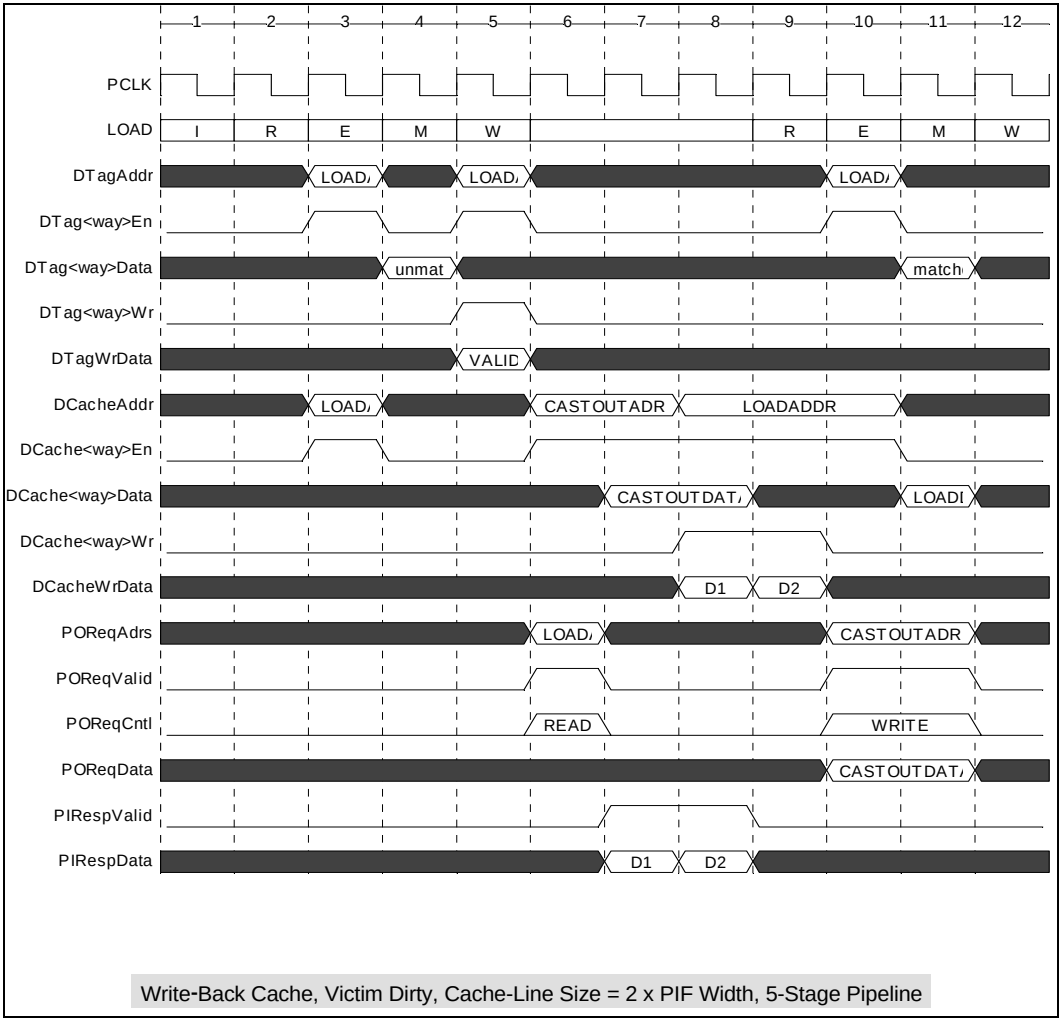


Figure 16–51. Cache-Load Miss (5-Stage Pipeline No Early Restart)

Above, the load operation misses so the line must be allocated to cache. Note that data return on the PIF in cycle 7, followed by cycle 8, represents the optimal case for this transaction. Additional cycle delays may be incurred if the store buffer contains items that need to be dispatched prior to a line refill.

During cycle 6, the processor asserts POReqValid, POReqAdrs, and POReqCntl to initiate a PIF read from external memory to satisfy the refill. The processor must also cast out a victim line. This second activity occurs in parallel with the load-miss line allocation. To cast out the dirty cache line, the processor asserts the castout address on DCacheAddr (cycle 6), resulting in the cache driving castout data on DCache<*way*>Data (cycles 7-8). This castout data is read over two cycles (linesize = 2 X PIF width) and stored in the PIF write buffer so that it can be sent out on the PIF (cycles 10-11). The load-miss activity continues with cycle 8, which shows the line being read from memory on PIRespData and the line being written to cache on DCacheWrData at cycles 8 and 9.

From cycle 9 onwards, the processor replays the load. (Note again that the instruction may be held in the R stage of the execution pipeline or in the P stage of the fetch pipeline if the load instruction cannot be re-fetched immediately.) The load address now matches the cache tag and the data is loaded via DCache<way>Data to complete the load. The castout activity is completed by casting out the least-recently-filled (LRF) cache line to external memory, as required for write-back cache. Cycles 10 and 11 show the castout address on POReqAdrs for a write of the corresponding data on POReqData.

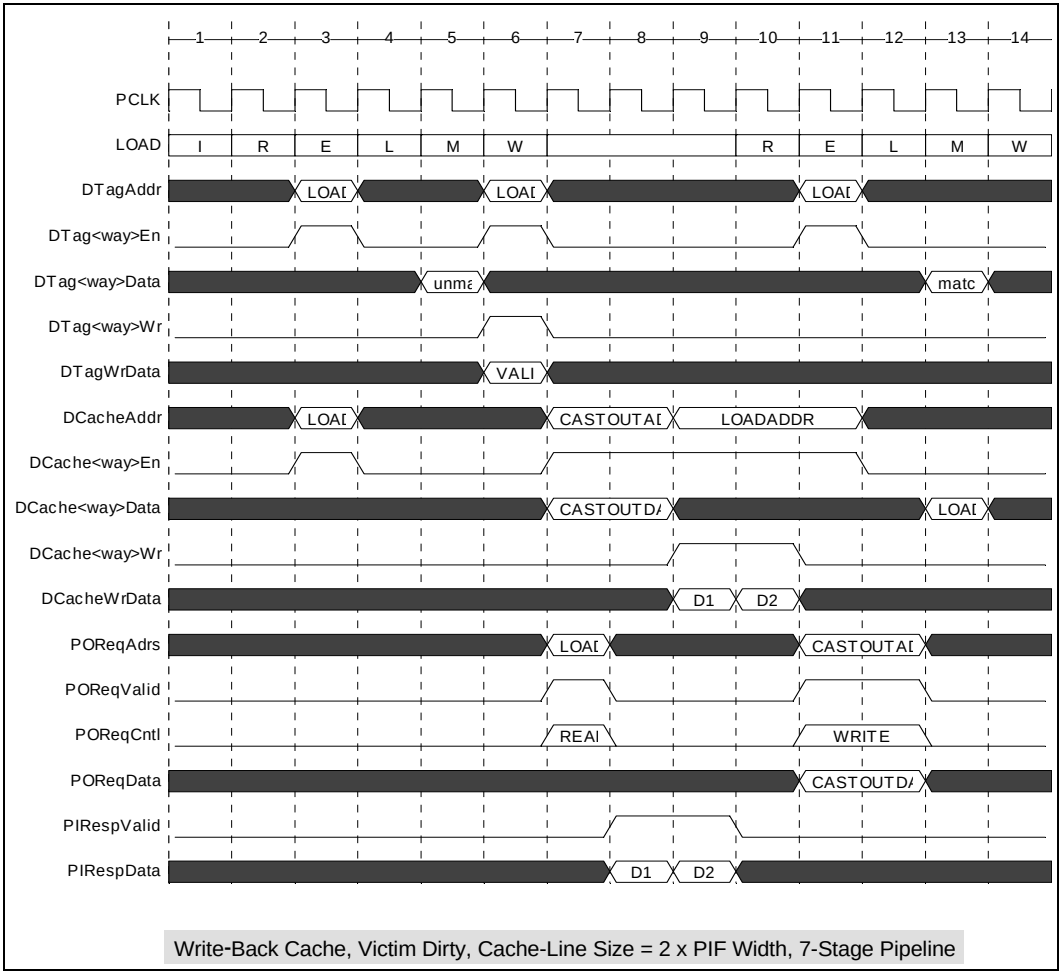


Figure 16–52. Cache-Load Miss (7-Stage Pipeline No Early Restart with Write-Back Caches)

In Figure 16–52 the load operation misses and the line needs to be allocated to cache. Note that data return on the PIF in cycle 7, followed by cycle 8, represents the optimal case in this transaction. Additional cycle delay may be incurred if the store buffer contains items that need to be dispatched prior to a line refill. During cycle 6, the processor asserts `POReqValid`, `POReqAdrs`, and `POReqCntl` to initiate a PIF read from external memory to satisfy the refill. The processor must also cast out a victim line, and this second activity occurs in parallel with the load-miss line allocation. To cast out the dirty cache line, the processor drives the castout address on `DCacheAddr` (cycle 7), resulting in the cache driving castout data on `DCache<`*`way`*`>Data`. This castout data is read over two cycles (linesize = 2 X PIF width) and stored in the PIF write buffer so that it can be sent out on the PIF (cycles 10-11). The load-miss activity continues with cycles 9-10, which shows the line being written to the cache on `DCacheWrData`. From cycle 11 on-

wards, the processor replays the load. (Note that the instruction may be held in the R stage of the execution pipeline or in the P stage of the fetch pipeline if the load instruction cannot be re-fetched immediately.) The load address now matches the cache tag and the data is loaded via `DCache<way>Data`, completing the load. The castout activity is completed by casting out the least-recently-filled (LRF) cache line to external memory, as required for write-back cache. Cycles 10-11 show the processor driving the castout address on `POReqAdrs` to write the corresponding data on `POReqData`.

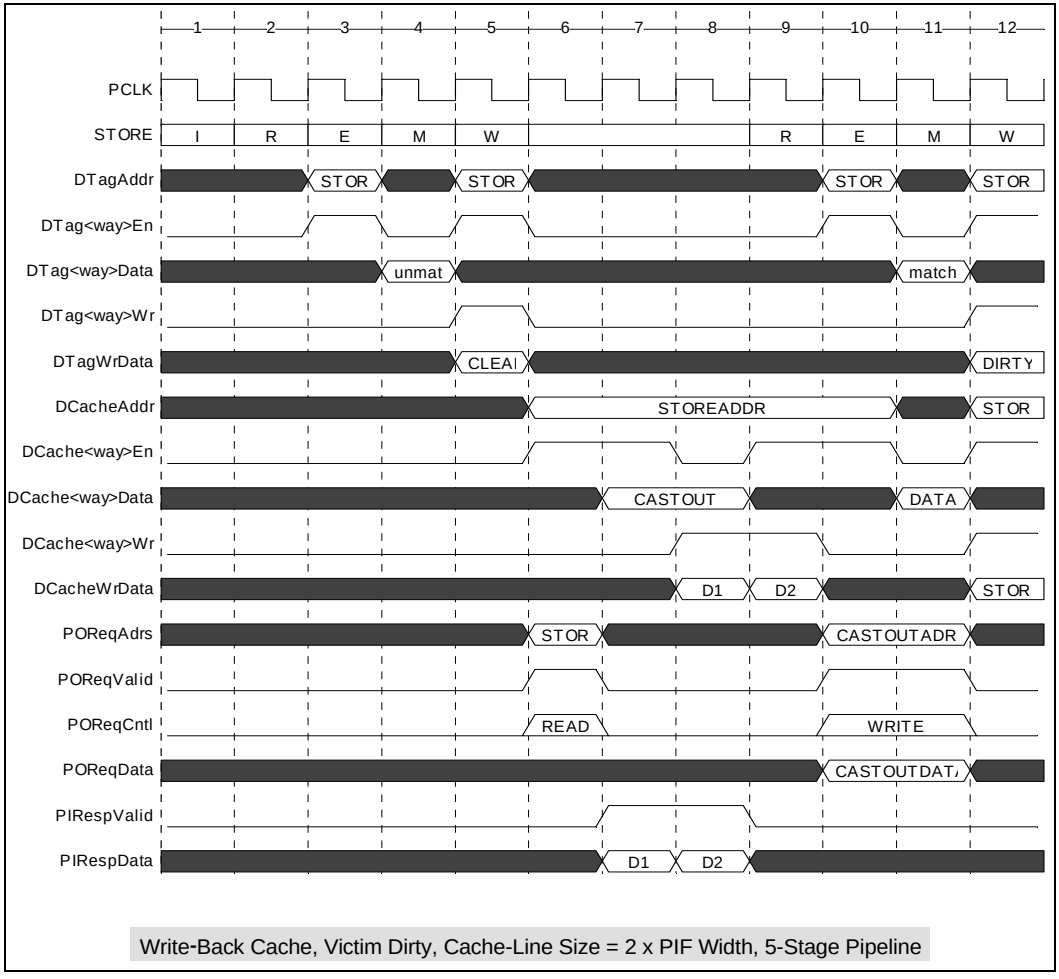


Figure 16–53. Write-Back, Cache-Store Miss, Victim Dirty(5-Stage Pipeline No Early Restart)

The above figure shows a cache-store miss where the line needs to be allocated to cache. Note that data return on the PIF in cycle 7, followed by cycle 8, represents the optimal case for this transaction. During cycle 6, the processor asserts `POReqValid`,

POReqAdrs, and POReqCntl to initiate a PIF read from external memory to satisfy the refill. The processor must also cast out a victim line, and this second activity occurs in parallel with the store-miss line allocation.

To cast out the dirty cache line, the processor asserts the castout address on DCacheAddr (cycle 6), resulting in the cache driving castout data on DCache<way>Data (cycle 7-8). The load-miss activity continues with cycle 8, which shows the cache line being read from memory on PIRespData and the cache line being written to cache on DCacheWrData at cycles 7-8.

From cycle 9 onwards, the processor replays the store. (Note again that the instruction may be held in the R stage of the execution pipeline or in the P stage of the fetch pipeline if the load instruction cannot be re-fetched immediately.) The store address now matches the cache tag and the data is stored via DCacheWrData. Note that in cycle 12, the tag array's dirty bit is set to indicate that the line has been modified.

The castout activity is completed by casting out the least-recently filled (LRF) cache line to external memory, as required for write-back cache. Cycles 10-11 show the processor driving the castout address on POReqAddr to write the corresponding data on POReqData. Although this figure illustrates that tag and data are updated simultaneously during cycle 12, these operations may not necessarily occur during the same cycle. The tag and data updates depend on the states of the tag buffer and store buffer, respectively.

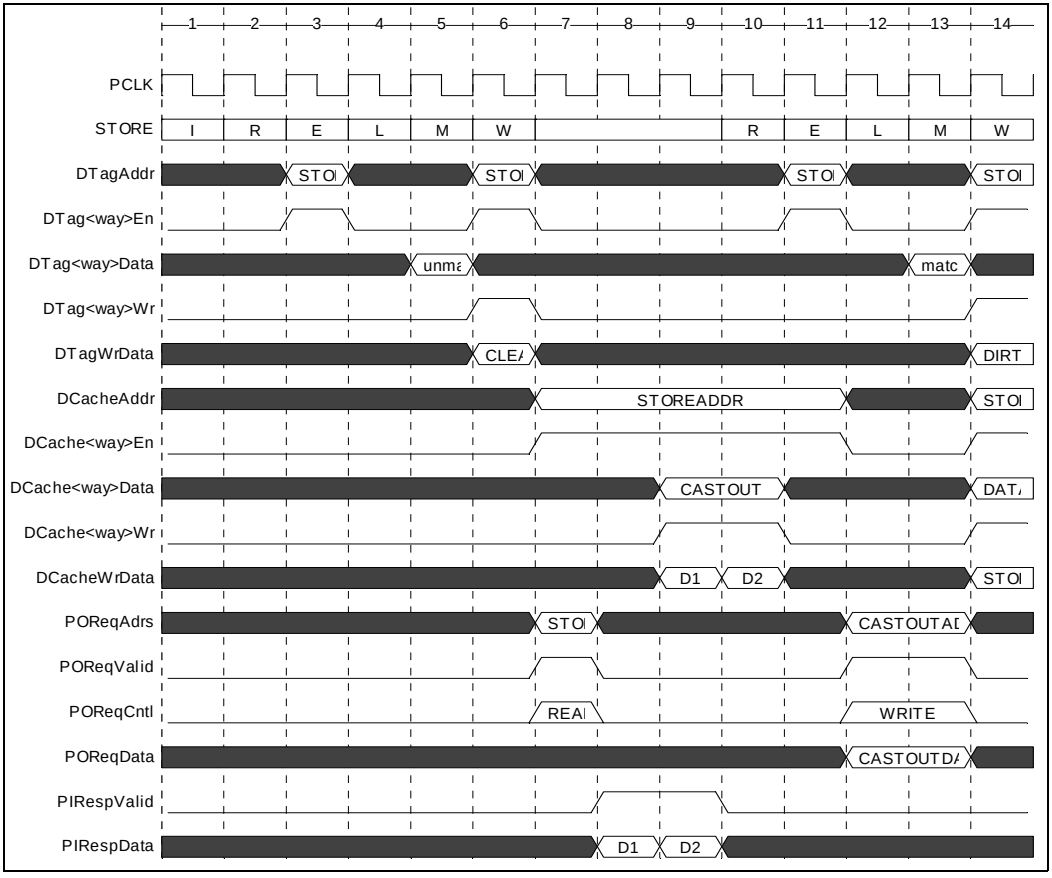


Figure 16–54. Write-Back, Cache-Store Miss, Victim Dirty (7-Stage Pipeline No Early Restart)

Figure 16–54 shows a cache-store miss where the line needs to be allocated to cache. Note that cycle 8, followed by cycle 9, represents the optimal case for this transaction. During cycle 7, the processor asserts `POReqValid`, `POReqAdrs`, and `POReqCntl` to initiate a PIF read from external memory to satisfy the refill.

The processor must also cast out a victim line, and this second activity occurs in parallel with the store-miss line allocation. To cast out the dirty cache line, the processor asserts the castout address on `DCacheAddr` (cycle 7), resulting in the cache driving castout data on `DCache<way>Data` at cycles 9-10. The load-miss activity continues with cycle 9, which shows the cache line being written to cache on `DCacheWrData` at cycles 9-10.

From cycle 11 onwards, the processor replays the store. (Note that the instruction may be held in the R stage of the execution pipeline or in the P stage of the fetch pipeline if the load instruction cannot be re-fetched immediately.) The store address now matches

the cache tag and the data is stored via DCacheWrData to complete the store. Note that in cycle 14, the tag array's dirty bit is set to indicate that the cache line has been modified.

The processor casts out the least-recently-filled (LRF) cache line to external memory, as required for write-back cache. Cycles 12-13 show the castout address on POReqAddr for a write of the corresponding data on POReqData. Although Figure 16–54 shows the tag and data being updated simultaneously, these operations may not necessarily occur during the same cycle.

16.5.1 Early Restart And Critical Word First

When early restart and critical word first are configured in the Xtensa core, the pipeline can be restarted as soon as critical data is returned from the PIF.

Following a load miss, the processor will replay and stall in the M-stage until the critical word is received. Meanwhile, the cache line refill can occur in the background, opportunistically writing the cache whenever there are free cache cycles available in the pipeline.

As a result of this behavior, the number of idle processor cycles following a miss can be substantially reduced as demonstrated in the following timing diagrams. Note: Early restart on store misses is not supported at the present time.

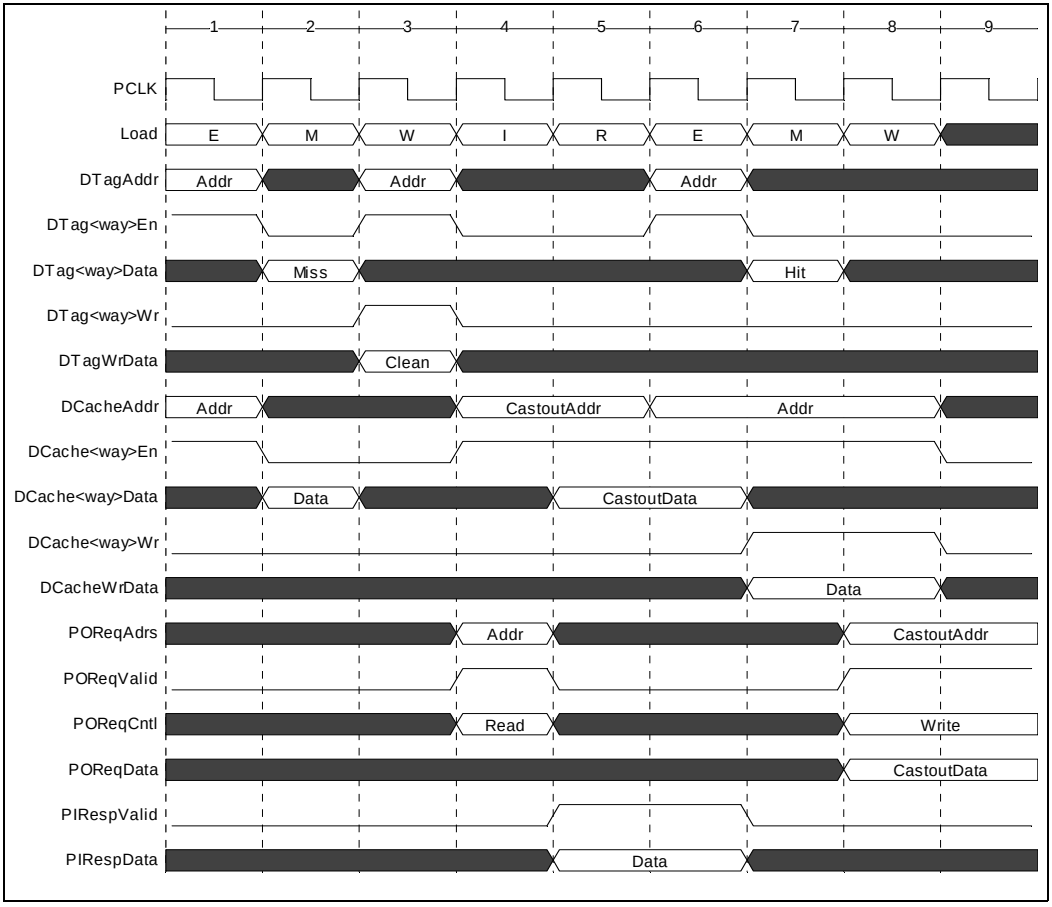


Figure 16–55. Load Miss 5 Stage Pipeline With Early Restart/Critical Word First

Figure 16–55 depicts a typical load miss scenario for a configuration with a 5-stage pipeline, writeback caches, and early restart/critical word first configured.

The primary difference for when early restart is configured is that the processor pipeline can resume as soon as the critical word arrives on the processor interface. In the example above, the tag is written “clean” at cycle 3 and the critical data from the miss arrives at cycle 5.

As a result, the critical data is bypassed to the load at cycle 7 and the pipeline does not stall.

Meanwhile in the background, the load operation misses, so the line must be allocated to cache. During cycle 4, the processor asserts `POReqValid`, `POReqAdrs`, and `POReqCntl` to initiate a PIF read from external memory to satisfy the refill.

The processor must also cast out a victim line. This second activity occurs in parallel with the load-miss line allocation. To cast out the dirty cache line, the processor asserts the castout address on DCacheAddr (cycle 4), resulting in the cache driving castout data on DCache<*way*>Data (cycles 5-6). This castout data is read over two cycles (line-size = 2 X PIF width) and stored in the PIF write buffer so that it can be sent out on the PIF (cycles 8-9).

In the background, the cache is refilled during cycle 7-8 while the pipeline is running.

Note: The cache refill is completely decoupled from the pipeline and will opportunistically fill the cache when free cycles are available. This timing diagram represents the best case scenario.

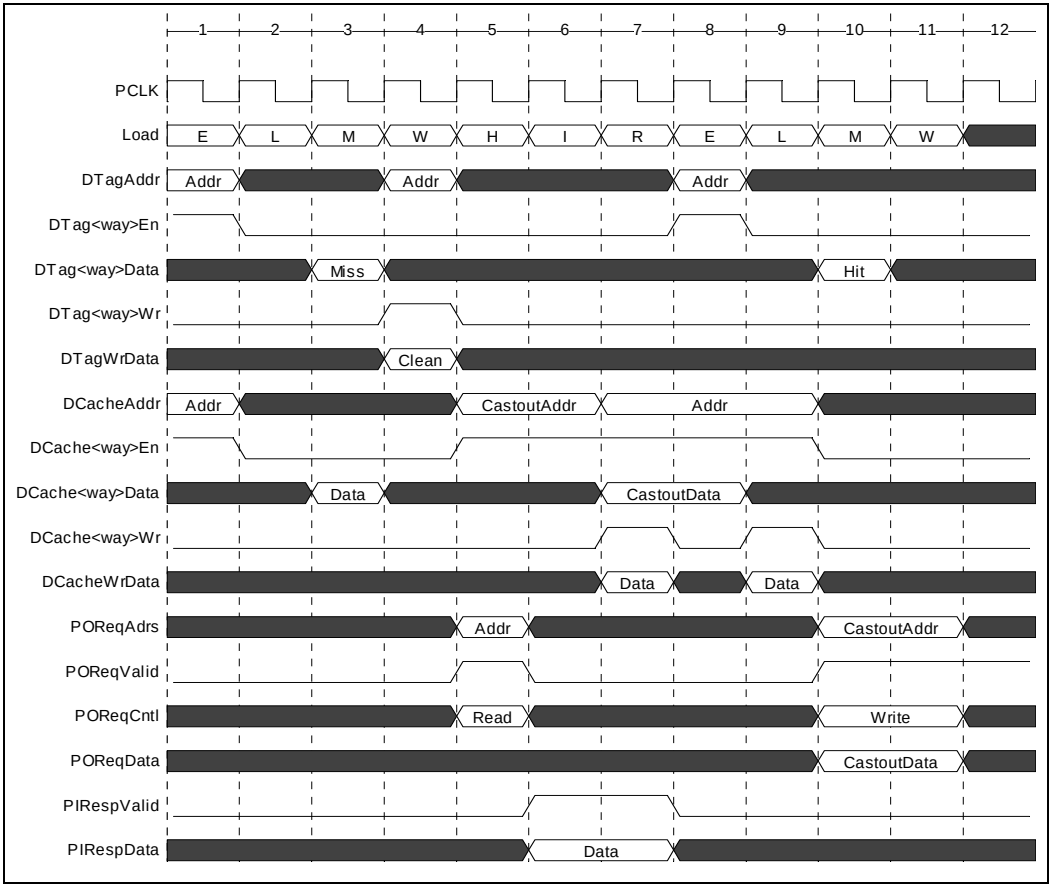


Figure 16–56. Load Miss 7 Stage Pipeline With Early Restart/Critical Word First

The 7-stage pipeline is similar to the 5-stage pipeline, except the data from the tag and data array arrive one cycle later. In the above diagram, the load misses in the cache and a read transaction goes out on the processor interface at cycle 5.

The read data arrives during cycle 6-7 and the critical data is bypassed to the load from the response buffer at cycle 10. As a result, the load commits at cycle 11 even though the cache refill does not complete until cycle 9.

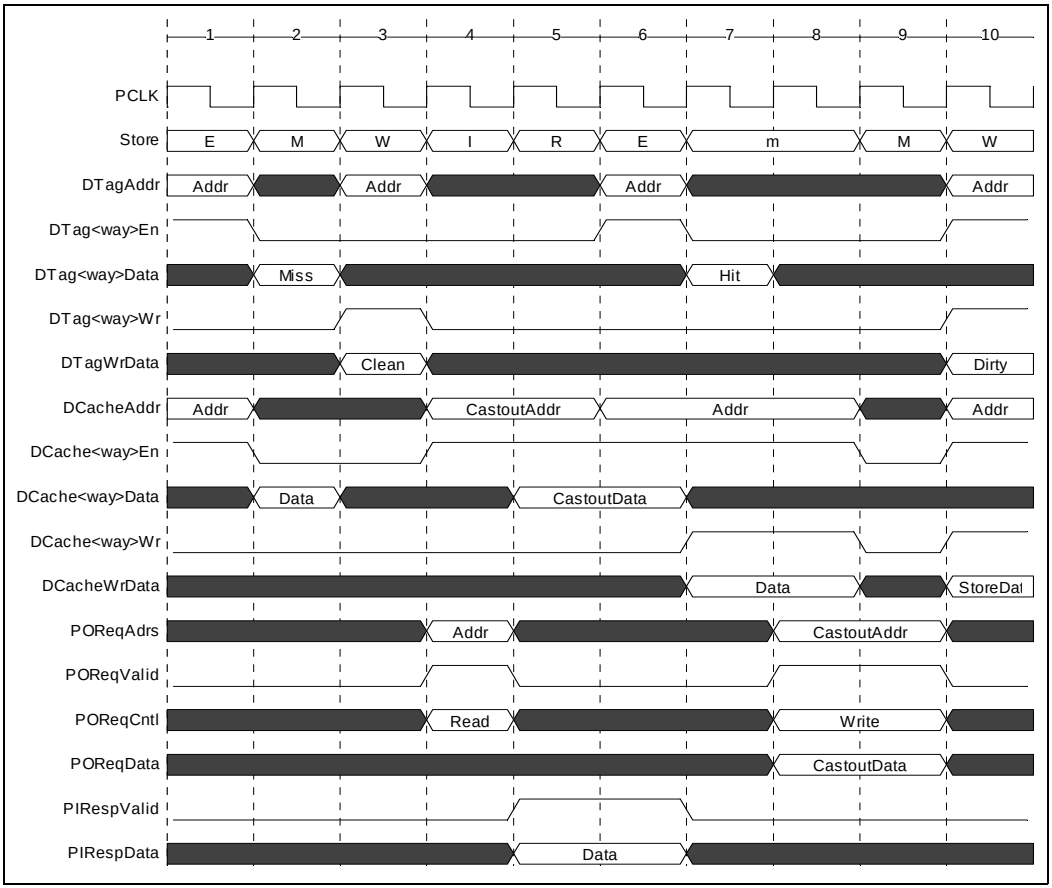


Figure 16–57. Store Miss 5-Stage Pipeline Early Restart

The store miss timing diagram with early restart is very similar to that of a processor without early restart with the following exceptions: For processors with early restart configured, the pipeline will stall in the M stage, as opposed to the R stage until the refill completes.

In the example above, a store miss results in a replay at cycle 3 where the line needs to be allocated to cache. Note that the data return on the PIF in cycle 5, followed by cycle 6, represents the optimal case for this transaction.

During cycle 4, the processor asserts POReqValid, POReqAdrs, and POReqCntl to initiate a PIF read from external memory to satisfy the refill. The processor must also cast out a victim line, and this second activity occurs in parallel with the store-miss line allocation.

To cast out the dirty cache line, the processor asserts the castout address on DCacheAddr (cycle 4), resulting in the cache driving castout data on DCache<way>Data (cycle 5-6).

The load-miss activity continues with cycle 7, which shows the cache line being written to cache on DCacheWrData at cycles 7-8. Meanwhile, the processor stalls in the M stage (cycle 7-8) until the cache refill completes.

Once the fill completes, the store address now matches the cache tag and the data is stored via DCacheWrData. Note that in cycle 10, the tag array's dirty bit is set to indicate that the line has been modified.

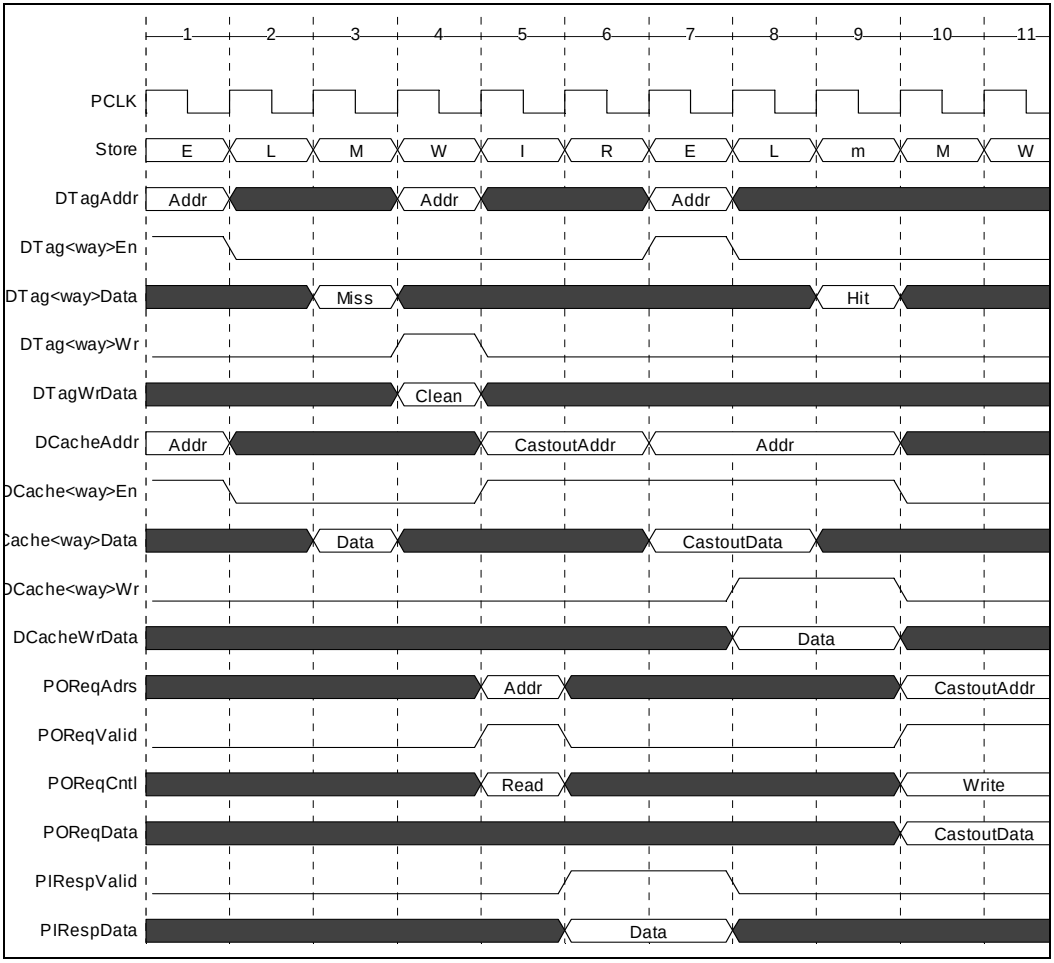


Figure 16–58. Store Miss 7-Stage Pipeline Early Restart

Likewise, a store miss in a 7-stage pipeline is similar to that of the 5-stage pipeline, except the memory latency is increased by an additional cycle.

In the example above, a store miss results in a replay at cycle 4 where the line needs to be allocated to cache. Note that the data return on the PIF in cycle 6, followed by cycle 7, represents the optimal case for this transaction.

During cycle 5, the processor asserts `PAReqValid`, `PAReqAdrs`, and `PAReqCntl` to initiate a PIF read from external memory to satisfy the refill. The processor must also cast out a victim line, and this second activity occurs in parallel with the store-miss line allocation.

To cast out the dirty cache line, the processor asserts the castout address on DCacheAddr (cycle 5), resulting in the cache driving castout data on DCache<way>Data (cycle 7-8).

The load-miss activity continues with cycle 7, which shows the cache line being written to cache on DCacheWrData at cycles 7-9.

The store address now matches the cache tag and the data is stored via DCacheWrData. Note that in cycle 11, the tag array's dirty bit is set to indicate that the line has been modified.

16.5.2 Data Cache Line Fills and Castouts

In most cases, a data-cache line fill cannot be canceled before it completes. Because all cache fills are decoupled from the pipeline and write the caches opportunistically, an exception or an interrupt will not affect a cache fill. Similarly, castouts can not be canceled once they have begun either.

In the event of a bus error, the tags are written as invalid once the fill of bogus data has completed.

16.6 Multiple Load/Store Units With Caches

Caches can be configured with processors with two load/store units. In order to maintain the maximum level of bandwidth within the core, the memories are organized in the following fashion when more than one load/store unit is configured.

- The tag memories are duplicated (one per load/store unit). All tag writes happen simultaneously, but each load/store unit can read its own local tag at any time
- The main cache memories are divided up into either 2 or 4 banks. For example, if an 8k cache is configured with 4 banks, each bank will be 2k in size and contain different pieces of the cache line. Note: The width of each bank is fixed at the width of the load/store unit.
- Each cache bank has its own set of memory interface pins. However, every way of the cache within the same bank shares the same address pins. As a result, a transaction to any given bank prevents other accesses to the same bank even if the destination is to a different way of the cache.

Depending on the lower bits of the memory address, each load/store unit will only enable the bank that is being accessed. Therefore, the chance that any given load/store unit will contend with the other is drastically reduced which should allow the processor to run at full bandwidth most of the time.

In the event that both load/store units issue loads to the same bank, the processor will stall to resolve the conflict. However, there are a few optimizations in place to speed up some common conflicts.

If the accesses from both load/store units match on the same load address, there is no stall. Instead, all bytes are enabled and the data from the cache memory is passed to both units. This allows for the sequential array accesses to occur simultaneously across load/store units without penalty.

Similarly, simultaneous store accesses from both load/store units to the same address are combined and issued as a single transaction. For example, if one load/store unit is doing a store to address 0x0 with byte enables set to 0x3 and the other load/store unit is doing a store to address 0x0 with byte enables set to 0xc, then only a single store will be issued to address 0x0 with byte enables set to 0xf.

16.6.1 Cache Banking

Cache banking allows for multiple masters to access the cache simultaneously as long as the accesses are to different banks. As a result, a cache with four banks can have up to four simultaneous masters.

16.6.2 Cache Access Prioritization

Besides the load/store units, there are many other sources that may contend for the caches. Below are a few scenarios regarding reads and writes from various sources and their relative priorities for access.

- All pipeline loads are always given priority over stores
- All pipeline loads and stores always have higher priority than cache refills, castouts and prefetches
- If both load/store units issue loads to the same bank, load/store unit 0 always gets priority access and the processor stalls until both load/store units have completed their cache accesses
- If both load/store units issue stores, then whichever load/store unit has more pending stores will receive higher priority
- All castouts and cache refill transactions have higher priority than prefetch refills
- Prefetch has the lowest priority access to the caches.

Cache banking is always performed based on the lower bits of the address. For example, if a load/store unit is 32 bits wide with 4 banks of cache, then bits[3:2] are used to select the bank that will be accessed.

16.6.3 Tag Accesses

With the requirement of maintaining duplicate tags across load/store units, all tags in the processor must be updated simultaneously in the order to maintain tag coherency. In the event that all tags are not available when a load/store unit wishes to write the tag, the tag write must be delayed until all tags are idle.

Because there can be multiple masters trying to write the tags simultaneously, the tag writes are prioritized as follows from (highest to lowest).

- Both load/store (LSU) units have the highest priority access to the tags. During cases where both load/store units miss at the same time, LSU0 always gets to write its tag first.
- Store tag buffer dispatches. When a dirty store occurs and both LSUs are trying to write their respective tags to the dirty state, whichever load/store unit has more dirty stores buffered up will have priority to write the tags.
- Bus error tag invalidates. When both load/store units are trying to do an invalidate at the same time, load/store unit 0always has priority.
- Tag updates due to prefetch transactions, which may affect the cache state

16.7 Cache Test Instructions

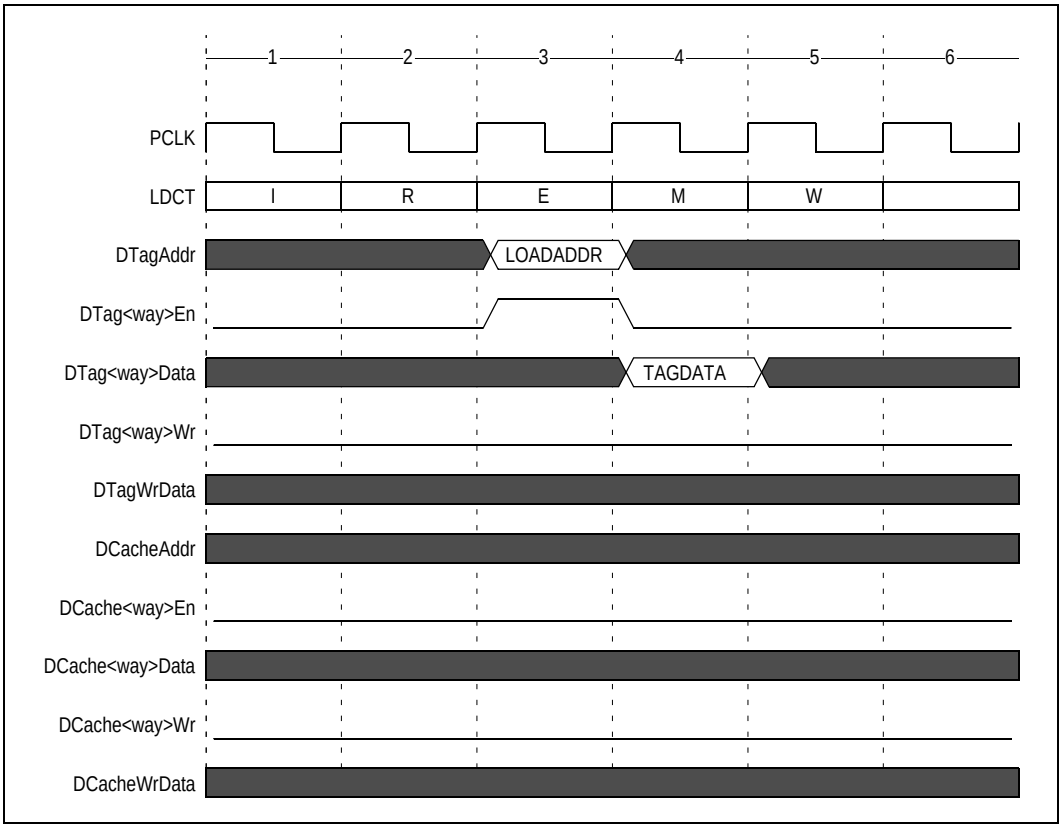


Figure 16–59. Load Data-Cache Tag Instruction (**LDCT**, 5-Stage Pipeline)

When there are two load/store units with caches, prefetch to l1 configured, both the LDCT and SDCT instructions behave slightly differently in order to perform a manufacturing test on the additional new tag memories.

Therefore, for LDCT and SDCT, the two bits just below the index will determine which "copy" of the tags will be read similar to how the two bits above the index say which "way" is accessed. SDCT will write all copies and LDCT will use these bits to read individual copies.

Note: Because the SDCT/LDCT instructions are meant for manufacturing tests only, the LDCT/SDCT instructions should only be performed when the processor is not handling any outstanding castouts. In addition, the prefetch option to L1 cache (if configured) should be turned off.

The encoding of this field will be as follows. A value of 2’d0 will refer to the load/store unit 0’s tag, a value of 2’d1 will be for the load/store unit 1’s tag and a value of 2’d2 will be for the prefetch tag.

The figure above shows how processor with a 5-stage pipeline loads the cache tag. During the E stage (cycle 3), the processor asserts `DTagAddr` and `DTag<way>En`, and reads `DTag<`*`way`*`>Data` during the load’s M stage. This figure shows the data cache index tag write operation for a processor with a 5-stage pipeline. All stores to the data cache tag occur in the W stage of the instruction. Note that the Xtensa processor’s LDCT and SDCT instructions only read or write one cache way while the DII instructions write all cache ways simultaneously.

Therefore, for LDCT and SDCT, the two bits just below the index will determine which "copy" of the tags will be read similar to how the two bits above the index say which "way" is accessed. SDCT will write all copies and LDCT will use these bits to read individual copies.

The encoding of this field will be as follows. A value of 2’d0 will refer to load/store unit 0’s tag, a value of 2’d1 will be for load/store unit 1’s tag and a value of 2’d2 will be for the prefetch tag.

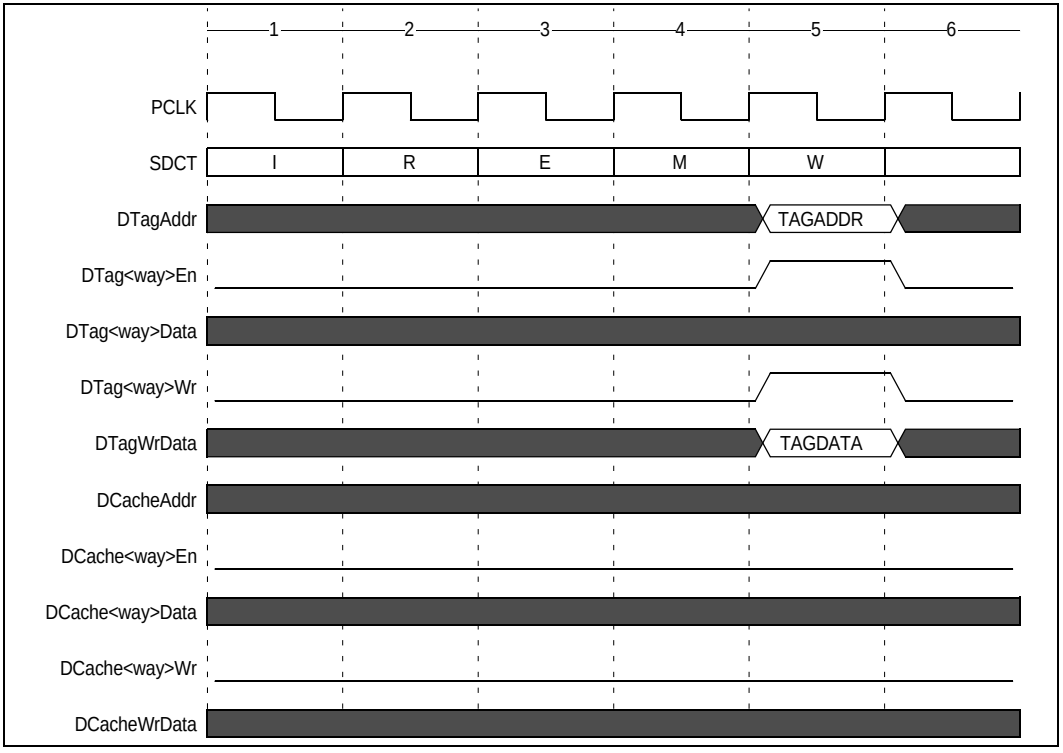


Figure 16–60. Data-Cache Index Tag-Writing Instructions (**`DII`**, **`DIU`**, **`SDCT)`**, 5-Stage Pipeline

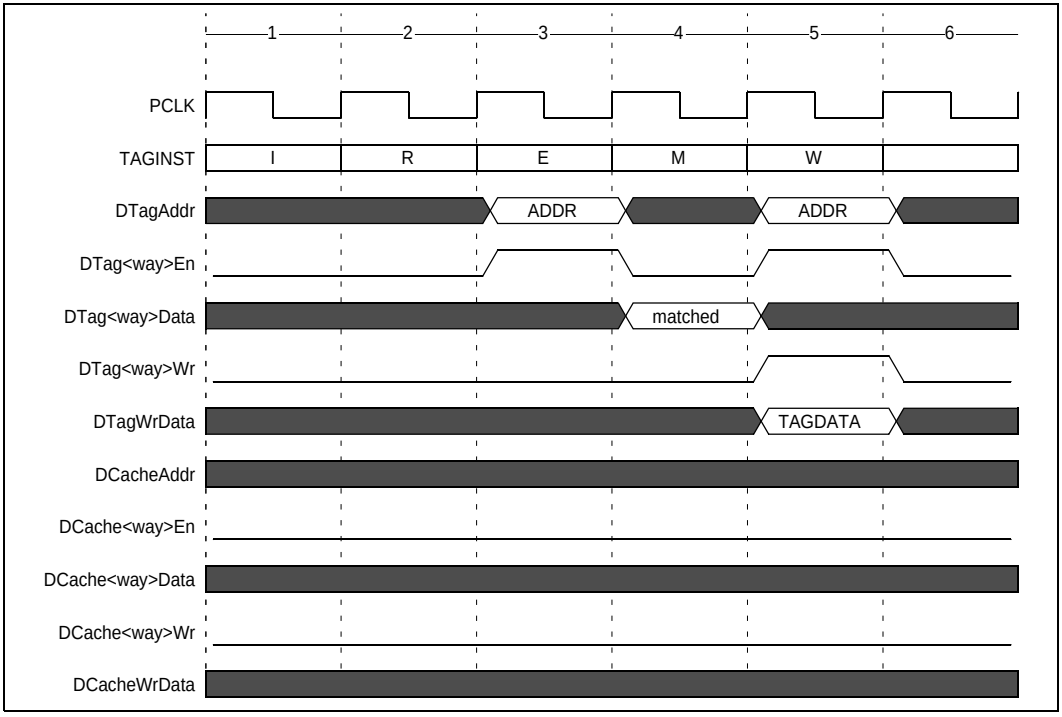


Figure 16–61. Data-Cache Tag-Hit Writing Instructions (**DHI**, **DHU**, 5-Stage Pipeline)

The figure above shows the operation of the DHI and DHU instructions in an Xtensa processor with a 5-stage pipeline. The data-cache tag-hit instructions become NOPs if the instruction's target address does not hit the cache. Otherwise, they update the tag in the W stage. The DHI instruction invalidates the cache tag and the DHU instruction unlocks the cache tag. All stores to the data-cache tag occur in the W stage.

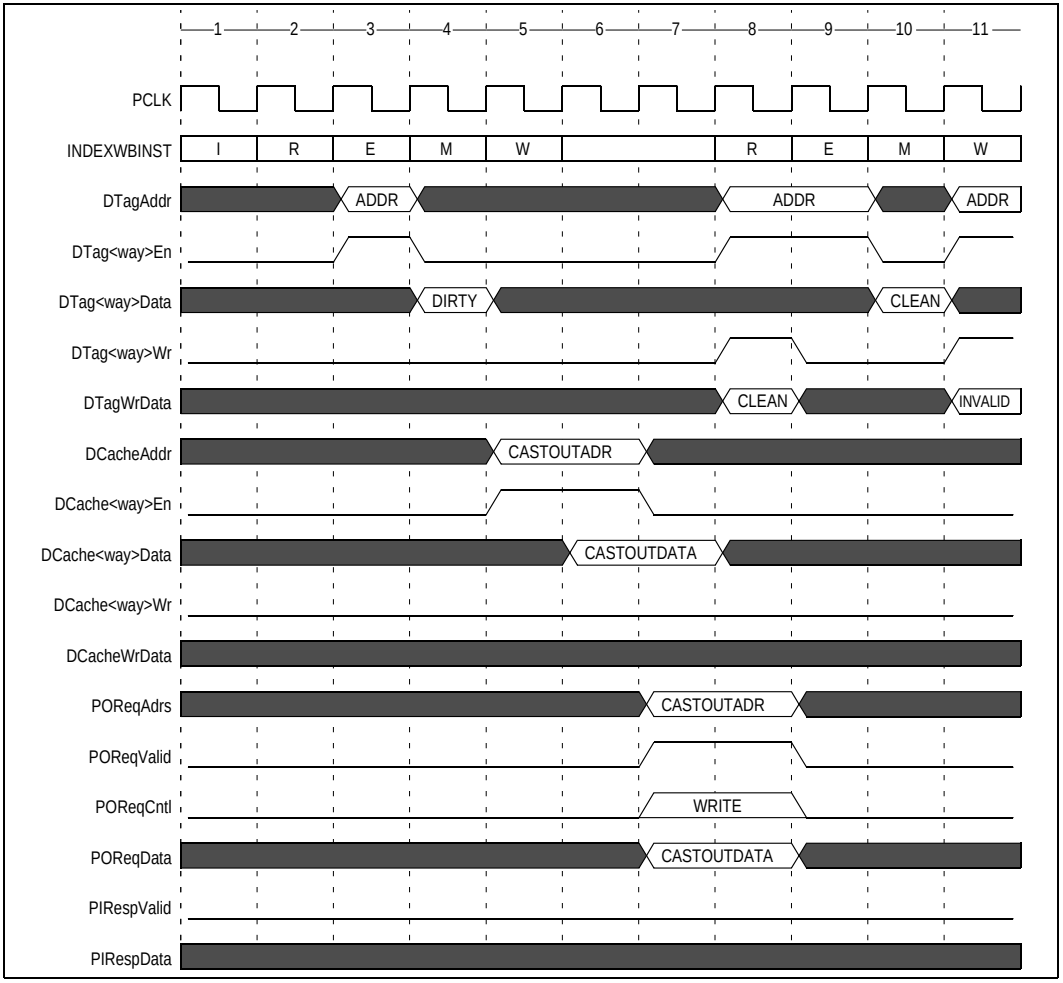


Figure 16–62. Data-Cache Tag-Writeback Instructions (**DIWB**, **DIWBI**, **DHWB**, **DHWBI**)

This figure shows tag-writeback instructions where the line is dirty and needs to be cast out. To cast out the dirty line the processor asserts the castout address on `DCacheAddress` at cycle 5. During cycle 7-8, the processor asserts `POReqValid`, `POReqCntl`, and `POReqAdrs` to initiate a PIF write to external memory. From cycle 8 onwards, the castout is completed and the processor replays the cache instruction.

The second time around, the tag is now clean and is ready to be invalidated in the case of the `DIWBI` or `DHWBI` instructions or left alone in the case of the `DIWB` or `DHWB` instructions. Note that if the line was initially clean, the waveform would show the same behavior as for a regular cache-tag instruction.

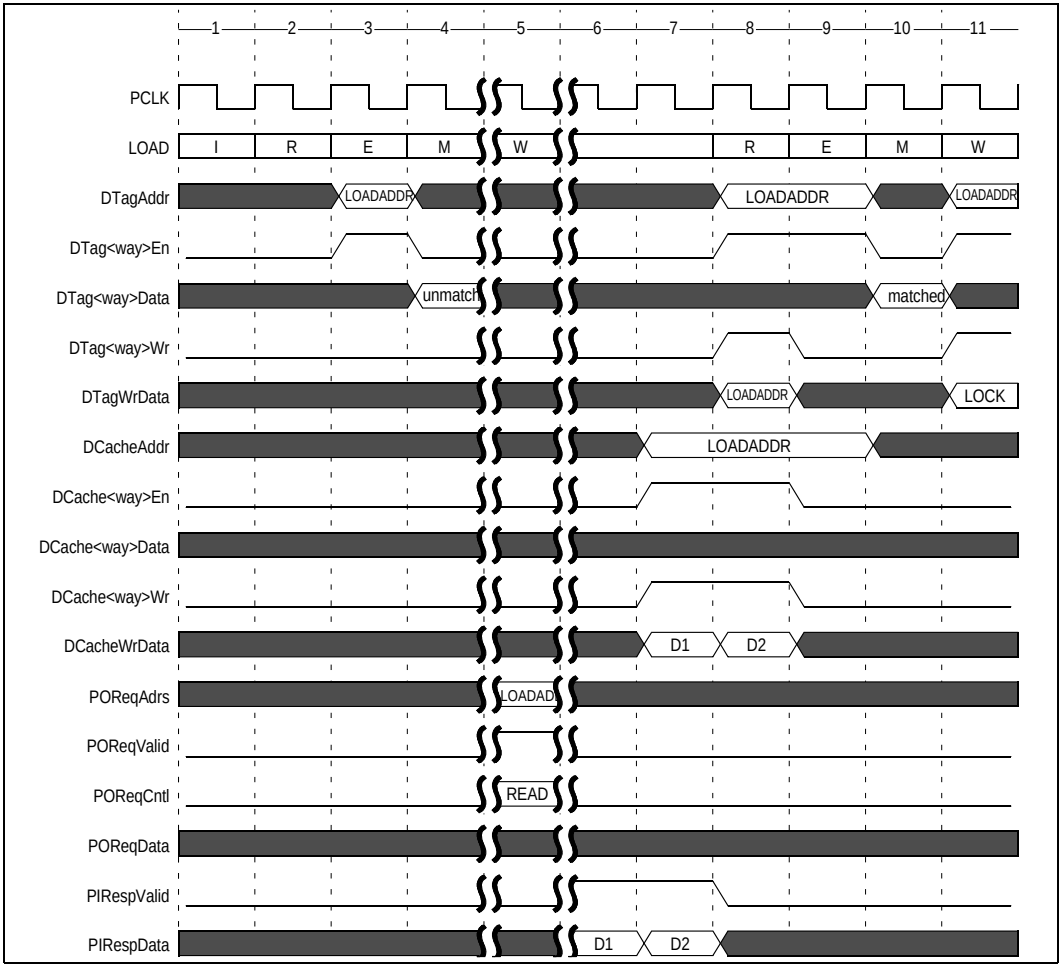


Figure 16–63. Data Prefetch and Lock Instruction (**DPFL**, 5-Stage Pipeline)

The figure above shows the operation of the DPFL instruction for an Xtensa processor with a 5-stage pipeline configuration. The DPFL instruction acts much like a load instruction. If the DPFL instruction misses in the cache, it initiates a PIF transaction to refill the cache line and the instruction is replayed. The second time around, the cache line is then locked. Note that a write-back cache may also require a cast-out operation.

16.8 Unaligned Data-Cache Transactions

When the Unaligned Handled By Hardware option is configured, the processor will automatically perform multiple transactions to sequential locations as needed to the data-cache interface to perform unaligned loads and stores. When an unaligned load or store

does not cross a data cache width boundary, only 1 transaction is required. However, when an unaligned access spans a data-cache-width boundary, the processor will perform the first portion of the transaction and will then replay to perform the second portion. Any data-cache access spanning a data-cache-width boundary will always require two transactions separated by a replay. Either or both of these two transactions could miss in the cache and thus could require additional replays to fill the cache line. In the worst case, an unaligned access could span two distinct cache lines, miss in both transactions and require three replays to complete.

When there are two load/store units with caches, prefetch to l1 configured, both the LDCT and SDCT instructions behave slightly differently in order to perform manufacturing test on the additional new tag memories.

16.9 Data Cache Conditional Store Transactions

The behavior of S32C1I conditional store transactions to the data cache depends on the value of the ATOMCTL register. The ATOMCTL special register controls how atomic operations behave within the Xtensa processor under different cache attributes. The purpose of this control is to restrict the generation of the special PIF bus transaction type called Read-Conditional-Write (RCW) in cases where external hardware is not able to handle that transaction type. In configurations with privileged instructions, changing the ATOMCTL register is a privileged operation.

Depending on the cache attributes and the ATOMCTL value, S32C1I will behave in one of the following ways:

- With ATOMCTL=Exception, the S32C1I instruction will take an exception once it reaches the W stage of the pipeline.
- With ATOMCTL=External RCW, the S32C1I instruction will behave in one of the four following ways.
    1. If a cache hit occurs to a clean line, the line is invalidated and the S32C1I is performed over the PIF using the special bus transaction type called Read-Conditional-Write (RCW).
    2. If a cache miss occurs, the S32C1I is performed over the PIF using the special bus transaction type called Read-Conditional-Write (RCW) and the cache is not updated.
    3. If a cache hit occurs to a dirty line, the relevant cache line is cast out and invalidated and the S32C1I is performed over the PIF using the special bus transaction type called Read-Conditional-Write (RCW).
    4. If a cache hit occurs to a locked line, the processor will take an exception.

- With ATOMCTL=Internal, the S32C1I instruction will be performed directly on the cache interface. If the S32C1I misses in the cache, the line will be brought in and the instruction executed without requiring any special PIF bus transaction types. S32C1I transactions to locked lines are permitted and behave as expected.
- S32C1I to isolate regions will take an exception.
- An unaligned S32C1I instruction will result in an exception.

16.10 Data-Cache Memory Integrity

The data cache can optionally be configured with memory parity or error-correcting code (ECC). These memory-integrity options cause additional bits to be read from and written to the cache data arrays and the cache tag arrays. The extra bits are used to detect or correct errors that may occur on the cache data. The parity option allows the processor to detect single-bit errors. ECC allows the processor to detect and correct single-bit errors and to detect double-bit errors.

If an uncorrectable memory error is detected on a data-cache access, an exception is taken if any byte of the cache data is used by the load which caused the access. If an uncorrectable memory error occurs in any of the tags during the data cache access, an exception is taken regardless of whether any of the data is used by the load. The exception occurs in the pipeline's W-stage for that instruction, just like any other exception.

Note: In general, most memory exceptions are precise. However, because castouts occur in the background asynchronous to the pipeline, uncorrectable errors on a castout transaction will cause an imprecise exception. When an imprecise exception is taken on a castout, no data is written out to the system and the castout is effectively cancelled.

When a correctable memory error is detected on a data-cache access, the processor's behavior depends on the `Data Exception` (`DataExc`) field of the `Memory Error Status Register` (`MESR`). If a correctable memory error is detected and the `DataExc` bit is set, the processor takes an exception. If a correctable memory error is detected and the `DataExc` bit is not set, the processor will correct the error in hardware and replay the instruction.

Following the replay, the data access will proceed using the corrected data. In some cases, both the tag and the cache data may contain correctable errors. For these cases, the processor will replay twice to correct the tag and cache data respectively before proceeding.

The processor behaves similarly for memory errors occurring on data-cache access instructions (see Section 16.9).

- `DHI,DHU,DHWB,DHWBI and DPFL`: These instructions will either replay or take an exception depending on the `DataExc` bit when a memory error occurs on any of the tags.

- `LDCT,DIWB,DIWBI,DIU`: These instructions read a single tag and will only take a memory error exception or replay depending on the `DataExc` bit if the indexed tag contains an error.
- `DII`: When the cache-locking option is selected, the `DII` instruction will either replay or take an exception depending on the `DataExc` bit when a memory error occurs on any of the tags. If the cache-locking option is not selected, `DII` will never cause a memory error.

Note: When the ECC logic corrects a memory error, the erroneous value is not corrected in memory. Only the value provided to the processor through the ECC logic is correct. The processor must write the corrected value out to the appropriate memory location to correct the value in memory.

16.11 Data Cache Byte Enables

Data Cache byte enables are valid for both loads and stores. For example, if the Data Cache has a 64-bit access width, on a load from Load/Store unit 0 that hits on a line in the data cache, the processor will only use the enabled bytes indicated by DCacheByteEn0[7:0]. Similarly for a store that hits in the cache, only the bytes enabled by DCacheByteEn0[7:0] should be written into the data cache array.

The TIE language allows load or store operations to enable or disable arbitrary bytes of the data load or store. To support store operations with arbitrary byte enables, the data cache array must be designed to write data only on the designated byte lanes and must not infer operand size or alignment from any combination of asserted byte enables.

17. Prefetch Unit Option

Microprocessor performance can suffer when main memory latency becomes very long and there are a significant number of data or instruction cache misses. The processor can spend many cycles just waiting for data or instructions that it needs. Prefetch is one mechanism for overcoming very long memory latencies that can occur in large System-On-Chip (SOC) designs. Assuming available bandwidth, data or instructions can be requested well ahead of time so that they are available when they are needed. In this way the penalty due to long memory latencies can be substantially alleviated.

There are several flavors of prefetch:

- Hardware prefetch automatically detects when a string of sequential data or instructions are being referenced and brought into the data or instruction cache, and requests the next set of data or instructions before they are needed.
- Software prefetch can be used to prefetch individual cache lines into the processor using special software prefetch instructions.
- Block prefetch is an extension of Hardware and Software Prefetch that allows processor instructions to specify an entire range of cache lines to be requested in the background.

The Xtensa processor offers a special Hardware Prefetch option which includes software prefetch instructions for individual cache lines, and can further extend this to a Block Prefetch option that allows blocks of memory to be operated on.

17.1 Prefetch Basic Operation

Figure 17–64 shows a simplified block diagram of Xtensa processor with the Prefetch option added.

Internally, the prefetch logic has a miss address table which keeps track of the instruction and data cache misses that have occurred, and prefetch entries that keep track of the cache line addresses that are being prefetched. The data that is being prefetched is stored in a memory external to the processor for area efficiency. There are a configurable number of prefetch entries (8 or 16) with each entry having cache lines worth of buffering in the external memory.

The prefetch engine has the option of either keeping the prefetched data in the prefetch buffers until it is requested by a cache miss, or of speculatively moving the prefetched data into the cache in order to try and avoid the cache miss entirely.

The prefetch engine looks at cache miss requests coming from the L1 cache logic, as well as cache load and store operations in the pipeline, and can detect incrementing sequential address streams. The prefetch engine can be configured to implement a limited number of prefetch strategies depending on a control register: For instance it can start prefetching once a sequential address stream has been detected, or it can be programmed to prefetch more aggressively and prefetch whenever a cache miss is detected.

In the case of requiring a stream to be detected before prefetching, a sequential access stream is detected whenever a cache line miss is detected for cache line address "a" and then a subsequent cache line miss is detected for the next cache line address "a+1". Once the stream is detected, cache lines at cache line addresses "a+2" and "a+3" are prefetched into the prefetch buffer. When an L1 cache misses on a cache line that is being prefetched, then data will come from either the prefetch buffer, or directly from the pending prefetch response. When the cache line "a+2" is brought into the L1 Cache, then the "a+4" cache line is requested in order to keep two prefetches active for this active stream. Note that the prefetch engine can optionally be bringing the cache lines into the L1 cache in the background.

- For more aggressive prefetching, the prefetch engine can be programmed to do a prefetch on any cache miss. For instance one prefetch policy is if a cache miss is detected for cache line address "a", then the prefetch engine will immediately start prefetching the next cache line address "a+1". If the cache miss hits in the prefetch entry, then it will prefetch cache line address "a+1" and "a+2".

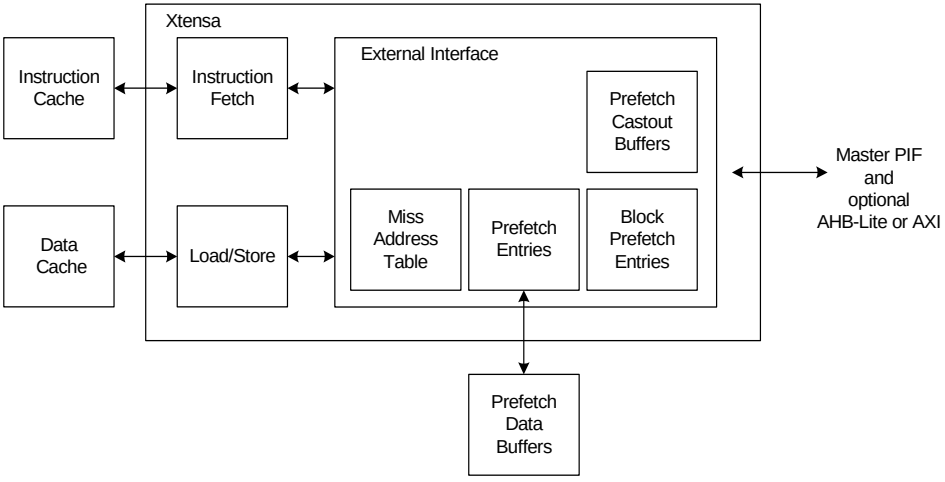


Figure 17–64. Basic Prefetch Block Diagram

17.1.1 Block Prefetch

The Prefetch Option can be extended to have Block Prefetch to the data cache. This allows Block Prefetch instructions to specify a range of addresses that are to be operated on based on a starting address and a size. The Block Prefetch logic will run through the entire range of addresses one cache line at a time, and perform the required prefetch operation. There are Block Prefetch operations that will do a traditional prefetch of a region into the L1 data cache, as well as Block Prefetch operations that will downgrade cache lines by invalidating them or casting them out. Block Prefetch also includes a Prefetch and Modify operation, also called Prepare to Store, which allows cache lines that are going to be completely overwritten to be made valid in the cache without requesting them over the processor bus, thus reducing latency, and requiring less bus bandwidth.

17.2 Prefetch Option and Architecture Description

The following are the important features of the Prefetch option:

- Allows the prefetch logic to be turned on or off, and supports different prefetch strategies with separate control for instruction cache and data cache prefetch.
- Can be configured to automatically detect sequential streams of data or instructions without program intervention or modification.
- Supports a configurable number of prefetch entries. A larger number of entries allows more streams of larger depth prefetch streams to be supported. For instance, eight prefetch entries can support four streams of depth 2, or two streams of depth 4.
- Optionally supports prefetching data directly into the L1 data cache. Note that this option requires an extra auxiliary tag array so that prefetch lookups can occur in parallel with pipeline cache accesses.
- Supports data cache software prefetch instructions (DPFR, DPFW, DPFRO, DPFWO).
- Hardware Prefetch does not prefetch across 64KB memory boundaries (this prevents unintended crossing of system memory boundaries during Hardware Prefetch). Note that block prefetch operations can cross a 64KB boundary.
- Implements the prefetch data buffers with a memory that is external to the Xttop level of the design hierarchy, similar to an instruction RAM, data RAM, cache memories, and TraceRAM. This is done for performance and area efficiency.
- Optionally supports Block Prefetch, allowing ranges of memory to be prefetched or castout from the data cache, including Block Prefetch Operations that prepare cache lines in the cache to be stored to without requesting them over the processor bus.

17.2.1 Configuration Options

The following are the Configurable Prefetch Options:

- Number of Prefetch Entries: 8 or 16.
- Prefetch to L1. Prefetch can be configured to allow prefetch strategies that bring prefetch data directly into the data cache in the background (Note: this option is not available for the instruction cache, which always keeps the prefetch data in the prefetch buffers, and brings it in only on an instruction cache miss.)
- Prefetch castout buffering: 1 or 2 cache lines. When Prefetch to L1 is configured, as well as write back caches, the prefetch logic may need to castout dirty lines from the cache. It does this in the background using the prefetch castout buffer, and this buffer can contain one or two cache lines worth of buffering.
- Block Prefetch: Requires the Prefetch to L1 option. There are eight Block Prefetch Entries, allowing eight block prefetch operations to be active at once. Note that a Block Prefetch Entry is different from a Prefetch Entry. Prefetch Entries are used by hardware prefetch, software prefetch, and block prefetch to fetch individual cache lines.

17.2.2 Configuration Restrictions

The Prefetch option has the following configuration restrictions:

- Configuration must have a data cache.
- A data cache line must be longer than one load/store access width.
- Prefetch to L1 requires that the Early Restart Option also be configured.
- The Block Prefetch option requires Prefetch to L1.
- The Block Prefetch option is not available with a full MMU, since in particular it requires the prefetch to L1 option, which is not compatible with a full MMU.

Prefetch for the instruction cache accesses will occur if the following restrictions are observed:

- Configuration must have an instruction cache in addition to a data cache.
- The data cache line size and the instruction cache line size must be the same.
- The data cache width and the instruction cache width must be the same OR the instruction cache width can be twice the data cache width with the added restrictions that the data cache width is the same as the PIF width, and the early restart configuration option is chosen.

If the above conditions apply, hardware prefetch will occur for instructions.

Note: Several DSPs, with wide SIMD, do not support instruction cache prefetch due to the third restriction such as the Vision P5, Vision P6, ConnX BBE32EP, and ConnX BBE64EP.

Note that instruction and data prefetch are separate in that a cache line that is prefetched because of hardware data prefetch or software data prefetch cannot be used by the instruction logic, and a cache line that is prefetched because of hardware instruction prefetch cannot be used by the Load/Store logic.

17.2.3 Implementation Restrictions

The Prefetch option has the following limitations:

- Only sequentially increasing streams are detected.
- A limited number of hardware prefetch strategies are implemented.
- Software prefetch is not available for the instruction cache (IPF instruction is not implemented)
- Prefetch directly to L1 strategies are not available for instruction caches.
- Block Prefetch is not available for instruction caches.

17.2.4 Prefetch Architectural Additions

Table 17–61 shows the Prefetch option architectural additions.

Table 17–61. Prefetch Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
PREFCTL	1	20	Prefetch Control	R/W[2]	40

1) Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions.

2) This register has special write properties.

The PREFCTL register (shown in Figure 17–65) controls whether hardware prefetch is turned on or off. Software prefetch instructions will still cause prefetch operations to occur even if hardware prefetching is turned off. Table 17–62 shows the PREFCTL register value meanings. For each of the instruction and data fields (InstCtl,DataCtl) only the values 0, 1, 3, 4, 5, and 8 are actually defined. Writing this register has the following special behavior:

- Writing 0 causes the field to be written to 0x0.
- Writing 1 causes the field to be written to 0x1.
- Writing 2 or 3 causes the field to be written to 0x3.

- Writing 4 causes the field to be written to 0x4.
- Writing 5 causes the field to be written to 0x5.
- Writing any value from 6 to 15 causes the field to be written to 0x8.

For example, writing 0x27 to PREFCTL will actually write 0x38.

The Block Control field (BlockCtl) controls the block prefetch functionality. Specifically, it controls the number of prefetch entries that are used to do individual cache line prefetch requests on behalf of a block prefetch operation. Only values 0x0 through 0x9 are defined, and writing this field with something larger than 0x9 will cause it to be set to 0x9.

Writing PREFCTL to 0 will also invalidate all the prefetch entries that are doing upgrades, including prefetch entries that have in progress memory requests. In the case of block prefetch, it will also invalidate all block prefetch entries that are doing upgrades.

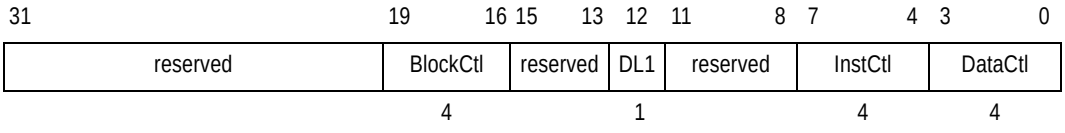


Figure 17–65. PREFCTL Register Format

Table 17–62. PREFCTL Fields

Field	Bit Index	Reset Value	Description
BlockCtl	[19:16]	0	Control Block Prefetch Upgrades (Note: if Block Prefetch Configured) 0: Block Prefetch Upgrades Disabled 1: Use maximum of 1 prefetch entry for block prefetch upgrades 2: Use maximum of 2 prefetch entries for block prefetch upgrades 3: Use maximum of 3 prefetch entries for block prefetch upgrades 4: Use maximum of 4 prefetch entries for block prefetch upgrades 5: Use maximum of 6 prefetch entries for block prefetch upgrades 6: Use maximum of 8 prefetch entries for block prefetch upgrades 7: Use maximum of 10 prefetch entries for block prefetch upgrades 8: Use maximum of 12 prefetch entries for block prefetch upgrades 9: Use maximum of 16 prefetch entries for block prefetch upgrades 10-15: Reserved
DL1	[12]	0	Data Prefetches should be put into L1 data cache if this is set (Note: This assumes that the Prefetch to L1 configuration option has been chosen. The cache line is not guaranteed to be put into L1 data cache, as certain conflict situations, such as the same cache way and index being used by a cache miss, can cause the prefetch to remain in the prefetch buffers).
InstCtl	[7:4]	0	Enables Instruction Prefetch according to the following encoding: ■ 0x0: Disabled ■ 0x1: Prefetch 1 cache line after stream is established ■ 0x3: Prefetch 1 cache line after every cache miss ■ 0x4: Prefetch 2 cache lines ahead when stream is established ■ 0x5: Prefetch 2 cache lines after stream is established, prefetch 1 for every other cache miss ■ 0x8: Prefetch 4 cache lines after stream is established, prefetch 2 for every other cache miss
DataCtl	[3:0]	0	Enables Data Prefetch according to the following encoding: ■ 0x0: Disabled ■ 0x1: Prefetch 1 cache line after stream is established ■ 0x3: Prefetch 1 cache line after every cache miss ■ 0x4: Prefetch 2 cache lines ahead when stream is established ■ 0x5: Prefetch 2 cache lines after stream is established, prefetch 1 for every other cache miss ■ 0x8: Prefetch 4 cache lines after stream is established, prefetch 2 for every other cache miss

Note that accessing the PREFCTL register when the Prefetch option is not configured will cause an illegal instruction exception.

17.2.5 Hardware Prefetch Strategies

The following prefetch strategies are supported for instruction and data caches. Note that in all cases only cached instruction and data miss requests cause prefetches to occur, and potentially get their data from the prefetch entries. A non-cached load will have no effect on the prefetch entries. Also, in the data cache case, the prefetched cache lines can be brought into the cache in the background based in the PREFCTL[12] bit (Note: in the case that prefetched cache lines are brought into the L1 data cache, a stream will be continued by loads or stores in the pipeline that hit in the cache as well as by those that cause cache misses):

1. PREFCTL[3:0]=0x1 or PREFCTL[7:4] =0x1: This strategy does not prefetch unless a stream has been established. A stream is established if a cache miss hits in the miss address table, of if it hits in the prefetch entry. One additional cache line will be fetched if the cache miss address hits in either the miss address table or a prefetch entry.
2. PREFCTL[3:0]=0x3 or PREFCTL[7:4]=0x3: This strategy will prefetch one cache line ahead on any cache miss.
3. PREFCTL[3:0]=0x4 or PREFCTL[7:4]=0x4: This strategy does not prefetch unless a stream has been established. A stream is established if a cache miss hits in the miss address table, or if it hits in a prefetch entry. Two additional cache lines will be fetched if the cache miss address hits in either the miss address table or a prefetch entry.
4. PREFCTL[3:0]=0x5 or PREFCTL[7:4]=0x5: This strategy will prefetch one cache line ahead on any cache miss that does not hit in a prefetch entry, and will fetch two cache lines ahead when a cache miss hits in a prefetch entry. The miss history table is not used.
5. PREFCTL[3:0]=0x8 or PREFCTL[7:4]=0x8: This strategy will prefetch two cache lines ahead on any cache miss that does not hit in a prefetch entry, and will fetch four cache lines ahead when a cache miss hits in a prefetch entry. The miss history table is not used.

Which strategy is best is configuration and application dependent.

17.2.6 Software Prefetch

The prefetch instructions DPFR, DPFRO, DPFW, DPFWO have been implemented. If prefetch to L1 is configured, then DPFR and DPFW will prefetch one data cache line into the L1 data cache regardless of the setting of the DL1 bit in PREFCTL; whereas DPFRO and DPFWO will prefetch one cache line into the prefetch buffer only, regardless of the setting of the DL1 in PREFCTL. If prefetch to L1 is not configured, then all these instructions fetch one cache line into the prefetch buffer. The IPF instruction cache prefetch in-

struction is implemented in the Xtensa processor as "no operation". The IPF opcode is recognized by the Xtensa processor, but it does nothing and will not cause any exceptions.

In order for the prefetch to occur, the address must be in a cached region, the cache line must not already be in the cache, and there must be a prefetch entry available for allocation (note that a prefetch entry is available for allocation if any entry is idle, or it is the oldest prefetch entry that is in a state where it has received all its data and does not need that data to do a refill or respond to a cache miss). The software prefetch instruction will end up being a NOP if it is to a no-allocate, bypass, or isolate region, or if it is to any memory that would not normally cause an L1 cache line fill such as a instruction or data RAM, an instruction or data ROM, or an XLMI port.

17.3 Block Prefetch Option

Configurations with Hardware Prefetch to the L1 data cache can be extended to also have a Block Prefetch feature. This allows memory address ranges to be brought into the cache, be castout from the cache, or be invalidated in the cache. The Block Prefetch instructions specify a starting address and a size, they allocate one of the Block Prefetch Entries, which will go though each cache line in the range and initiate the required action. The types of actions that are possible are:

- Normal Prefetch Operations: All cache lines in the address range will be prefetched into the L1 data cache.
- Downgrade Operations: All cache lines in the address range will be castout or castout and invalidated, or just invalidated, depending on the specific instruction.
- Prefetch and Modify (also known as Prepare to Store) Operations: All cache lines that are completely contained within the address range will be marked dirty in the cache without requesting data from main memory. Cache lines at the beginning or the end of the block may not be completely contained in the address range, and if that is the case then these cache lines are just prefetched normally.

Normal Block Prefetch Operations make cache line requests to the prefetch logic and use the same hardware that is used to do Hardware Prefetch or Software Prefetch of cache lines.The PREFCTL register specifies the maximum number of prefetch entries that can be used to handle Block Prefetch cache line requests. This allows some of the prefetch entries to also be used for Hardware Prefetch and Software Prefetch operations. Note that the entries are not statically reserved for block prefetch, and can be used by hardware and software prefetch if not being used by block prefetch requests.

Downgrade Operations and Prefetch and Modify Operations use two special prefetch entries, since they do not need to request data from memory. Note that these are in addition to the configured number of Prefetch Entries.

Several Block Prefetch Operations can be in progress at the same time, up to the number of Block Prefetch Entries configured. A grouping mechanism is provided whereby some ordering of the Block Prefetch Operations can be enforced. Specifically, Block Prefetch Operations are part of the current group, unless they specify that a new group is to be started. Earlier groups have priority, and will complete before the next group starts. The groups are processed in a FIFO manner. Within a group, different Block Prefetch Requests are handled in a roughly round-robin manner, so that they all make progress at the same time. Note that general Prefetch and Modify and downgrade type of operations will progress faster since they do not need to make requests to main memory, though they may require a cache line to be castout from the cache.

Table 17–63 shows the Block Prefetch Instructions that have been implemented in Xtensa LX7. There are instructions to help create new Block Prefetch groups, to wait for Block Prefetch completion, and cancel in progress Block Prefetch Operations. Note that these instructions cause an exception in the case that Block Prefetch is not configured.

Table 17–63. Block Prefetch Option Instruction Additions

Instruction	Definition
DPFR.B	Block Prefetch For Read to D-cache (current implementation requests cache lines exclusive, but it could request shared in the future)
DPFR.BF	Begin new group, and Block Prefetch For Read to D-cache (current implementation requests cache lines exclusive, but it could request shared in the future)
DPFW.B	Block Prefetch For Write to D-cache (requests cache lines exclusive)
DPFW.BF	Begin new group, and Block Prefetch For Write to D-cache (requests cache lines exclusive)
DPFM.B	Block Prefetch and Modify (also called Prepare to Store). Mark all cache lines completely in the block as dirty. Cache lines that are not fully contained in the block will be prefetched normally.
DFPM.BF	Begin new group, and Block Prefetch and Modify (also called Prepare to Store). Mark all cache lines completely in the block as dirty. Cache lines that are not fully contained in the block will be prefetched normally.
DHWB.B	Block Writeback data cache by writing back dirty cache lines
DHWBI.B	Block Writeback data cache by writing back dirty cache lines and invalidating them
DHI.B	Block Invalidate data cache by invalidating cache lines without writing back dirty cache lines. Cache lines that are not fully contained in the block will be castout and invalidated.
PFNXT.F	Start the Next Block Prefetch Group
PFWAIT.R	Wait for all required Block Prefetch operations (all downgrades) to be finished. This instruction will stall in the R-stage of the pipeline until all required Block Prefetch Operations have completed.
PFWAIT.A	Wait for all Block Prefetch Operations to be finished. This instruction will stall in the R-stage of the pipeline until all Block Prefetch Operations have completed.
PFEND.O	End all Optional Block Prefetch Operations (all upgrades)
PFEND.A	End all Block Prefetch Operations

Note that Block Prefetch upgrade operations, including Prefetch and Modify (Prepare to Store) operations, are considered optional. That is, they are not required for correct program execution, but are there purely to improve performance. In some cases cache line requests that are made on behalf of a Block Prefetch upgrade may be dropped due to conflicts of various sorts.

However, Block Prefetch downgrade operations (DHWB.B, DHWBI.B, DHI.B) are considered required, since after completion of the operation one expects that all cache lines in the range will have been castout and/or invalidated depending on the instruction. In particular, if the address range is to be castout, once the operation completes and all the write responses have been received, the data must be valid in the external memory.

17.3.1 Block Prefetch Restrictions and Limitations

Block Prefetch Operations have certain restrictions and limitations, as follows:

- Upgrade Block Prefetch instructions (DPFR.B, DPFR.BF, DPFW.B, DPFW.BF, DPFM, DPFM.B) will not allow prefetch operations to cross a 512MB address boundary since memory attributes may change at that point. Prefetch will be done up to the boundary and then stop.
- Downgrade Block Prefetch instructions (DHWB.B, DHWBI.B, DHI.B) will take an exception if the address range crosses a 512MB address boundary.
- Normal Block Prefetch instructions (DPFR.B, DPFR.BF, DPFW.B, DPFW.BF) are limited in size to 2X the data cache size. If the size is larger than 2X, prefetch operations will occur on the first 2X of the address range, and then stop. Note that having the size larger than the cache may be useful in applications that consume data, as it comes in the cache and then no longer needs the data; however, this may be hard to exploit in general as it is timing and event dependent (e.g., an interrupt may cause us to no longer be consuming the data).
- Prefetch and Modify (also known as Prepare to Store) Block Prefetch instructions (DPFM, DPFM.B) are limited in size to 1X the data cache size. If the size is larger than 1X the data cache size, Prefetch and Modify operations will occur on the first 1X of the address range, and then stop.
- Downgrade Block Prefetch instructions (DHWB.B, DHWBI.B, DHI.B) are limited in size to 1X the data cache size. If the size is larger than 1X, then a Load/Store exception will be taken.

Block Prefetch instructions have different behaviors depending on the attributes and properties of the addresses that they are block prefetching, as shown in Table 17–64.

Table 17–64. Block Prefetch Behavior Based on Memory Attributes and Type

Instructions	Illegal Memory Region	Cached Memory Region	Uncached Memory Region	In Local Memory (Data RAM/Data ROM)
`DPFR.B, DPFR.BF, DPFW.B, DPFW.BF`	Exception	Do upgrade	NOP	Do upgrade for all cache lines not in the local memory
`DPFM.B, DPFM.BF`	Exception	Do upgrade	NOP	Do upgrade for all cache lines not in the local memory
`DHWB.B, DHWBI.B, DHI.B`	Exception	Do downgrade	Do downgrade	Do downgrade for all cache lines not in the local memory

17.3.2 Block Prefetch Operation Pipeline Conflicts

Block Prefetch Operations may conflict with Loads and Stores in the pipeline. For instance, a store may go to a cache line that a block Prefetch and Modify (also known as Prepare to Store) operation will eventually perform a Prefetch and Modify operation on. Ordering must be preserved to ensure correct operation. The processor pipeline will replay or hold instructions due to certain conflicts with the Block Prefetch operations:

- Load Instruction: Replay if it overlaps with a remaining range of DPFM.B, DPFM.BF, DHWBI.B, or DHI.B operations.
- Store Instruction: Replay if it overlaps with a remaining range of DPFM.B, DPFM.BF, DHWB.B,DHWBI.B, DHI.B operations.
- DHI, DHWBI instructions: Replay if they overlap with a remaining range of any Block Prefetch Instruction.
- DPFL, DHWB, DHU instructions: Replay if they overlap with a remaining range of DPFM.B, DPFM.BF.
- DPFR, DPFRO, DPFW, DPFWO instructions: Replay if they overlap with a remaining range of DHWBI.B, DHI.B.
- DII, DIWB, DIWBI, DIU, DIWBUI.P instructions: Stall in the R-stage until all Block Prefetch operations have completed.

17.4 Additional Prefetch Implementation Details

17.4.1 Prefetch Stream Detection and Prefetch Entry Allocation/De-allocation

Although many things can affect prefetch operation, the basic stream detection and prefetch entry allocation works as follows.

The miss address table is used to detect the start of a prefetch stream in some of the prefetch strategies. There are 8 miss address tables entries. The entries are allocated in FIFO order whenever an instruction or data cache miss also misses in the prefetch entries.

There are a configurable number of prefetch entries (8 or 16), and each entry has a cache line's worth of buffering associated with it. The prefetch entries are either free, or are allocated and have an age based on their allocation order. If there is a free prefetch entry, then it is allocated, and set as the youngest prefetch entry. If there are no free prefetch entries, the oldest one is allocated to the new prefetch provided it has fully received its response from memory. Otherwise, no prefetch entry is allocated.

An individual prefetch entry is de-allocated under various conditions:

- When it finishes responding to an actual instruction or data cache miss.
- Whenever a store address going out on the PIF matches the prefetch entry address.
- Whenever a Store-Compare-Conditional (S32C1I) instruction address matches the prefetch entry address.
- When a prefetch entry receives a response from the PIF that has a bus error.

All the prefetch entries are de-allocated under various conditions:

- The PREFCTL register is written to 0.
- One of the cache invalidate instructions is executed (see Section 17.5.1).

Note that if a prefetch entry is de-allocated when it is waiting for a response from the PIF, it must wait for the response to complete before it can be allocated to another prefetch operation.

17.4.2 Prefetch Memory

The prefetch memory has the following characteristics:

- Width: If the Xtensa Early Restart Option is configured, the prefetch memory width will be maximum (data cache width, 2X the PIF width). That is, the prefetch memory will either be as wide as the data cache if the data cache is wider than the PIF, or it will be two PIF widths if the data cache width is the same as the PIF width. Having this restriction allows the prefetch logic to both accept prefetch responses from the PIF at full bandwidth, and read the prefetch memory to respond to a cache miss or to fill the L1 cache. If the Xtensa Early Restart Option is not configured, then the prefetch memory will be the same as the data cache width. Note that instruction and data caches must have the same width for the instruction prefetch to be available.
- A cache line’s worth of data for each prefetch entry.
- Always a single cycle memory, where the address and control are is presented on one cycle and the data is returned on the next cycle.

Even for a configuration with a 7-stage pipeline where there is extra pipeline registers between all the local instruction and data memories and the processor, the prefetch memory will not have these registers.

For example, a configuration that has early restart configured, a 64-bit wide PIF, 64-bit wide caches, a 64-byte cache line, and eight prefetch entries, the prefetch memory will be a 128-bit wide, 512-byte memory, with a 5-bit address input.

17.4.3 Prefetch Buffer Memory Errors

Beginning in Xtensa LX7, error checking can be configured for prefetch buffer memory. Legal options include ECC, Parity, and none. Parity coverage is 1 parity bit per 32 data bits. ECC is 7- ECC bits per 32-data bits.

Error checking on the prefetch buffer memory comprises a simpler implementation than those of the other memories in the design. This is due to the fleeting and replicate nature of the data contained therein. Where a data cache could possibly contain the only valid copy of data for a given address, the prefetch buffer may not. A prefetch memory line holds speculative data that may or may not be used. It is also read only once and then left in an invalid state.

Parity or ECC is generated and checked for writes and reads to the prefetch RAM during normal operations similar to the local memories. However, unlike local memories, there are no instructions to read or write the prefetch RAM. It is assumed that this memory will be tested using a built-in self test (BIST).

The error handling scheme performs the following for any detected error:

- Invalidates the prefetch buffer line that contains the bad data
- Invalidates any cache line tag or any internal load/store or Ifetch response buffer that this data may have begun to propagate to
- For data prefetch read errors, replays any instruction that is attempting to use the bad data from the prefetch read that detected the error until it hits again in the response buffer or data cache
- For instruction prefetch read errors, replays the instruction address that first attempted to use the bad prefetch data until it hits again in the Ifetch response buffer or instruction cache
- Signals the outside world (via two primary output signals) as to whether this was a single- or double-bit error (for ECC) or an error was detected on a single output (for parity)

No exception is taken, so essentially all of these errors are "correctable" in so much as the errant data is thrown out unused and re-fetched from somewhere other than the bad copy that was just invalidated in the prefetch buffer memory. These errors are not logged in the processor register.

The primary outputs are listed in Table 17–65. These signals are one cycle pulses. Note that in this table "Uncorrected" and "Corrected" imply what might have happened in a traditional ECC or parity error detection block. Since all of the data is thrown away and no exception is taken, these signals are really differentiating between Parity/double-bit ECC and single-bit ECC errors.

Table 17–65. Prefetch Buffer Memory Primary Outputs

Output	Description
PrefetchRamErrorUncorrected	Pulsed upon detection of any valid parity or double-bit ECC error.
PrefetchRamErrorCorrected	Pulsed upon detection of any valid single-bit ECC error.

17.4.4 Prefetch To L1 Data Cache

Data cache prefetch can be configured to put the prefetch responses into the L1 data cache in the background (Note that prefetch to L1 is not available for the instruction cache). Prefetch to the L1 data cache adds a considerable amount of complexity and area to the design and has a number of possible advantages and disadvantages. The performance benefits are highly system and application dependent.

- Checking for presence of cache line in the data cache: before making the prefetch request on the PIF, the prefetch logic will check whether the cache line is already present in the data cache or not. This involves reading and checking the tags. This will delay the prefetch going out to the PIF, but can save PIF bandwidth if the cache line is already in the cache.
- Effect on data cache miss rate: Prefetching into the L1 data cache can reduce cache misses if the prefetch has time to make it into the L1 data cache before the first reference to this cache line. However the prefetch refill logic can also replace a cache line that is still needed, and thus increase cache misses.
- Prefetch to L1 has a substantial area cost: There is an extra tag array that allows the prefetch logic to check for the presence of a cache line without interfering with the pipeline execution. If write back caches are configured, the prefetch logic needs to be able to castout dirty cache lines in the background, and this requires a prefetch castout buffer of 1 or 2 cache lines.
- Due to cache line allocation conflicts, prefetch to L1 is not guaranteed to put every prefetch into the data cache. In the case of certain cache line allocation conflicts with the Load/Store units, the prefetch logic can choose to keep the prefetch response data in the prefetch buffer, and not bring it into the L1 data cache. For these cases the cache line will be brought into the L1 data cache from the prefetch buffer when the actual data cache miss occurs. In the case that this occurs for a prefetch request that is due to a Block Prefetch operation, the line will be dropped, and the prefetch entry will move on to work on the next cache line in the block.

17.5 Prefetch Interactions with Other Components

17.5.1 Cache Operations and Prefetch

The following cache invalidate instructions will cause all certain prefetch entries to be invalidated:

- IHI, III (if instruction prefetch is present): invalidates all instruction prefetch entries
- DHWBI, DHI: invalidates a data prefetch entry if the address matches
- DII, DIWBI, DIWBUI.P: invalidates all the data prefetch entries
- IIU (if instruction prefetch and index locking is present): invalidates all the instruction prefetch entries

17.5.2 Memory Ordering and Prefetch

None of the ordering instructions will consider any prefetches for the purposes of ordering. This includes `MEMW`, `EXTW`, `EXCW`, and `ISYNC`. However since some of these operations must wait for write responses, castouts due to prefetch will be temporarily discontinued.

Upon executing a `WAITI` instruction, the PWaitMode output signal will not be asserted until there are no outstanding PIF requests, and waits until all prefetch and block prefetch operations have completed.

17.5.3 Interrupts and Prefetch

An interrupt can abort a cache miss request. When the miss history table is allocating multiple prefetch entries and an interrupt occurs, the prefetch entry allocations in the same cycle are allowed to complete, but all subsequent allocations will be canceled. Prefetch operations that are in-flight will continue as usual.

17.5.4 PIF and Prefetch

The Prefetch option will allow a number of outstanding PIF prefetch requests up to the number of prefetch entries. This means there can be many more simultaneous outstanding PIF requests than a configuration without the Prefetch option. System designers must take this into account and allow these requests to proceed in parallel to benefit from the Prefetch option. In general, the following properties apply to PIF requests and responses with respect to prefetch:

- Prefetch Requests: There can be the number of prefetch entries and each entry can have its own PIF prefetch request. These requests are in addition to the usual read and write requests. Prefetch requests to the PIF have the lowest priority (lower than PIF requests) due to load, store, and castout operations.
- Prefetch Address Memory Boundaries: Prefetch requests to the PIF will not cross a 64KB boundary. This helps reduce hardware cost and prevents requests from crossing undesired system memory boundaries.
- Prefetch Responses with Bus Errors: A bus error on the prefetch response will cause the prefetch entry to be invalidated. Note that a data error can occur on any word of the response, and the entire response is still received. If the prefetch entry is forwarding its response to an actual cache miss, then the bus error will be forwarded to the cache fill logic, and a bus error exception will be taken.
- Prefetch and PIF Attributes: If the PIF Request Attribute is configured, prefetch requests will go out with a specific PIF Attribute (as explained in Section 13.3.12 “Bus Errors”, “If no interrupt is allocated for write responses, then write transactions are assumed complete when an external agent acknowledges the processor’s write request with the PIReqRdy signal. The system designer may reserve an external interrupt pin to signal bus errors generated by write transactions. The external agent can assert this external interrupt, and the interrupt handler then decides how to handle the bus error.") Note that this attribute distinguishes between an instruction prefetch request, and a data prefetch request (Note that a software prefetch is a data prefetch request).

17.6 Basic Prefetch Waveforms

The following waveforms show a simple example of a stream being detected via the miss history table, and the timing of the Prefetch PIF requests that go out as a result.

Figure 17–66 shows how a prefetch stream is initially detected in the case that Prefetch to L1 is not configured:

- In cycle 2, a request for address "a" is received from an L1 Cache miss on the instruction or data cache, and it misses in the prefetch entries. The request is issued in the next cycle, which is the same request latency as configurations without prefetch. A history table entry is allocated with the next incremented line address, shown as "History0".
- In cycle 7, a request for address "a+1" is received, which hits in the history table, and misses in the prefetch entry. This triggers the scheduling of prefetches of the next two lines of addresses.
- In cycle 8, address "a+2", incremented from the history table, is looked up in the prefetch entries. During this operation no new requests are accepted.
- In cycle 9, address "a+3" is looked up in the prefetch entries. It is allocated if not already present. During this operation, no new requests are accepted.

- In cycle 9, the pending prefetch entry for address "a+2" is detected and its request is scheduled for the next cycle. Prefetches have lower priority compared to new PIF requests.
- In cycle 11, there is a prefetch bubble. There is a prefetch bubble issued after every prefetch that is issued on the PIF. A different request could have been issued in this slot.
- In cycle 11, there is a prefetch bubble cycle, meaning that another prefetch read request is not issued in this cycle. There is a prefetch bubble cycle after every prefetch read that is issued on the PIF. A PIF request not due to prefetch, such as a PIF request due to a cache miss, load, or store operation, could be issued in this cycle if it is available.

Figure 17–67 shows how a prefetch stream is initially detected in the case that Prefetch to L1 is configured. The only difference is that now we always check whether the cache line is present in the L1 cache before requesting it on the PIF. This can save bandwidth in the case that the line is present, but this checking delays the prefetch requests on the PIF by three cycles compared to Figure 17–66.

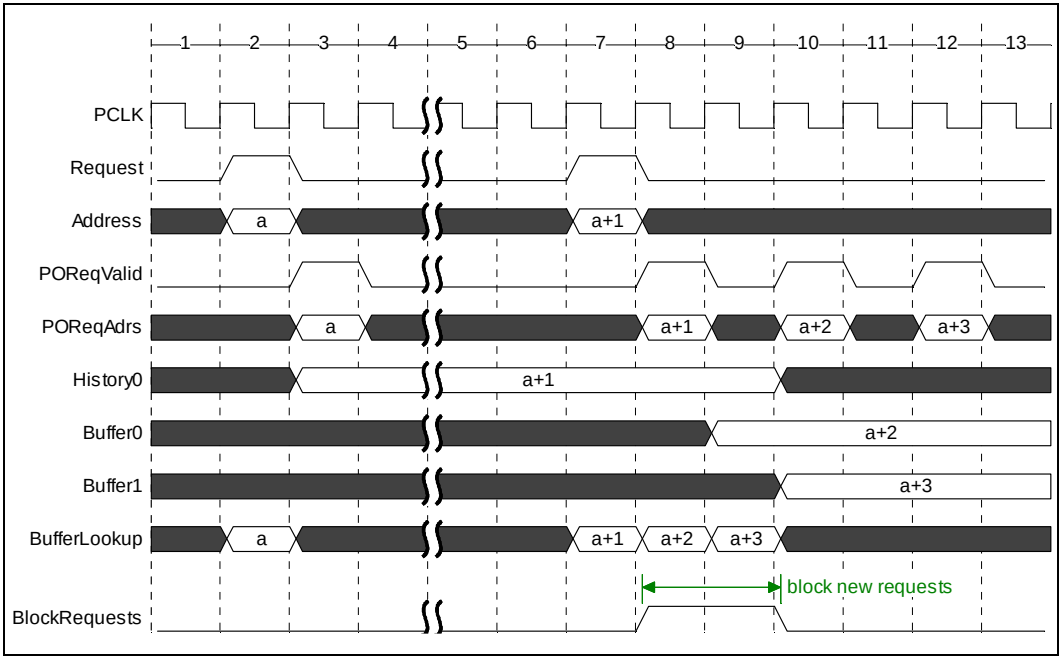


Figure 17–66. Detecting the Beginning of a Prefetch Stream, Prefetch to L1 not Configured

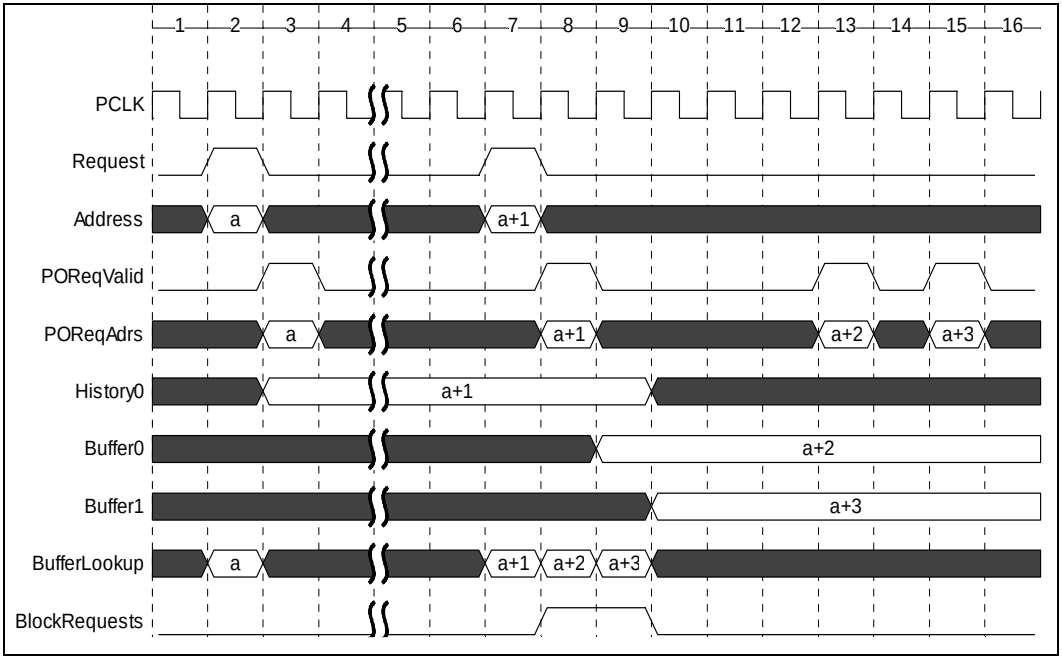


Figure 17–67. Detecting the Beginning of a Prefetch Stream, with Prefetch to L1 Configured

Figure 17–68 shows how data is returned from the prefetch entry, and how the prefetch stream continues once it is detected.

- In cycle 2, a request for address "a+2" is received, and this cache line has already been fetched into the prefetch buffer.
- In cycle 3 a lookup of the address "a+3" is performed. If "a+3" misses, it would be allocated. If "a+3" hits, as in this example, "a+4" is checked once the prefetch entry used for "a+2" is free.
- In cycle 4, the prefetch buffer is enabled with the address of the first word of the line. In this case the prefetch memory is twice the data cache width, so only two fetches are required to fetch the entire cache line (words i, ii, iii, and iv).
- In cycle 5, the first data from the prefetch memory comes back and is registered. This will be forwarded to the pipeline on the next cycle.
- In cycle 6 the prefetch entry used for a2 is free, so we do the lookup and allocate of a4.
- In cycle 7, the first refill request can be sent to the L1 cache. This refill will go on in the background. As shown there is no contention for the L1 cache and the refill completes in 4 cycles.
- In cycle 8, a+4 is requested on the PIF

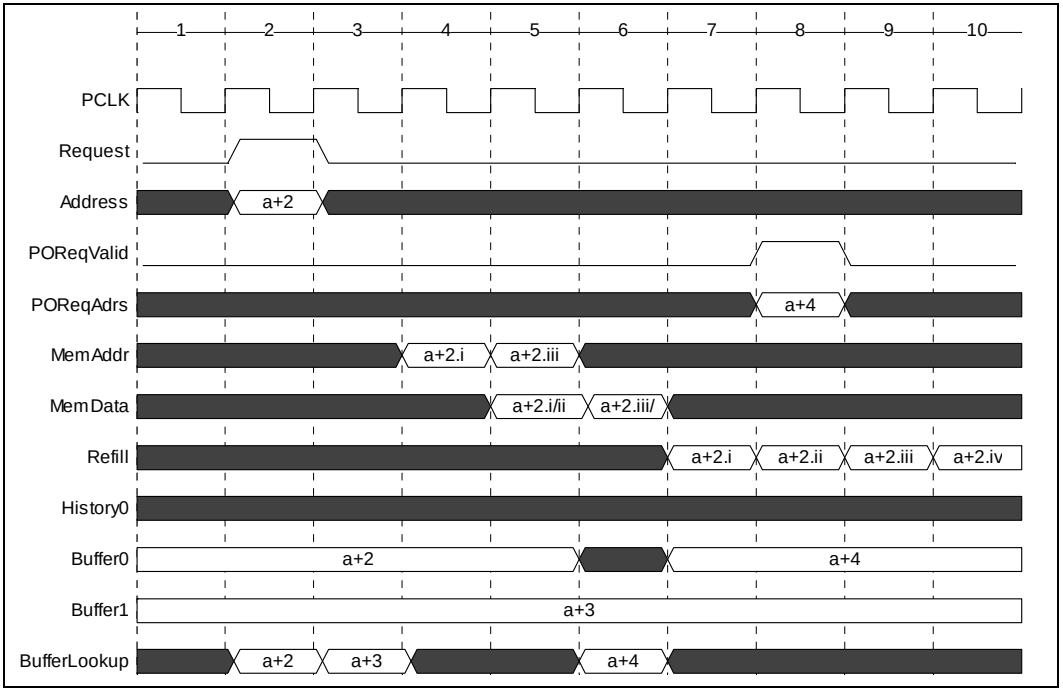


Figure 17–68. Returning Data from a Prefetch Buffer and Continuing the Prefetch Stream

17.6.1 Prefetch Scenarios

The waveforms shown are of very simple scenarios, but in a real system there can be many different timings of requests and responses depending on the all the simultaneous activity in the processor and system. The processor attempts to maximize the bandwidth used at the PIF and cache level. The following are some notes regarding different behaviors:

- The cache miss corresponding to a prefetch can occur before the prefetch response has completely arrived. The prefetch response will be either forwarded directly to the pipeline and refill logic, or may have to complete its fill of the prefetch buffer and then read the data from the prefetch buffer.
- In the case of Prefetch to L1, a cache line needs to be allocated to prefetch line coming back, and this may require the castout of another cache line. This castout occurs in the background, and can delay the refill to L1.
- In the case of Prefetch to L1, the cache miss corresponding to the prefetch may occur while the prefetch logic is in the middle of doing the L1 refill. In this case the pipeline lets the refill complete, and then gets the data from the cache.

- A data cache prefetch can temporarily be put in the prefetch buffer because it is waiting for an L1 cache line to be allocated, or because the prefetch logic is filling another cache line to the L1 cache, and then fill to the L1 cache once it is possible to do so.
- In the case of Prefetch to L1, there are a number of situations that cause the refill to L1 to be canceled. One example is that an unrelated cache miss uses the same cache line for its refill. In this case the prefetch data will remain in the prefetch buffer, and can be taken from there if it is needed.
- Many different agents are arbitrating for the PIF, the prefetch entry lookup and allocate, and the cache tag and data arrays. Many responses and requests can be outstanding and in progress, leading to complicated interactions and timing scenarios.

17.7 Software and Simulation Support for Prefetch

Prefetching is fully supported in software and in the instruction set simulator.

17.7.1 Controlling Prefetch in Software

Cache prefetching is controlled via the PREFCTL special register. On reset, prefetching is disabled, but standard Cadence LSPs using XTOS enable it in the reset vector.

You can use HAL calls to explicitly enable or disable prefetch from sections of your code. For example:

```
#include <xtensa/hal.h>

xthal_set_cache_prefetch( XTHAL_PREFETCH_ENABLE );

or

xthal_set_cache_prefetch( XTHAL_PREFETCH_DISABLE );
```

Many more constants are available to more finely specify the requested changes to PREFCTL, such as prefetch aggressiveness for instruction and data caches, whether to prefetch directly to L1 cache, and so on. For more details, please refer to descriptions of the `xthal_set_cache_prefetch()` and `xthal_get_cache_prefetch()` functions in the *Xtensa System Software Reference Manual*, specifically in the *Xtensa Processor HAL* chapter, *Cache* section, and *Global Cache Control Functions* subsection.

17.7.2 Prefetch in the Instruction Set Simulator

When running the instruction set simulator, you can use the `--prefetch` command-line option to set the PREFCTL register for the entire simulation run to a specific value without any code changes. For example, `--prefetch=0x44` sets PREFCTL to 0x44, `--prefetch=0` sets PREFCTL to 0, and so on.

```
xt-run --mem_model --prefetch=0 ...
```


17.8 Choosing Prefetch Configuration Options

The optimal choice of prefetch options and strategies is application and system dependent. The following are some general guidelines to think about when choosing the number of prefetch entries, and whether prefetch to L1 should be configured, and whether Block Prefetch is useful:

- Number of Prefetch Entries: This depends on the number of streams and the depth of streams that the application can benefit from based on the application and expected memory latency. If multiple streams are required, then more prefetch entries are needed. If the memory latency is very long then the prefetch depth may need to be higher, again requiring a larger number of prefetch streams. If the application can easily generate many software prefetch requests then more prefetch entries may be required.
- Prefetch to L1 versus No Prefetch to L1: If prefetch to the L1 data cache is configured then the application may be able to avoid certain data cache misses altogether, and better use the L1 data cache bandwidth, and thus improve performance. On the other hand prefetching directly to L1 may replace certain cache lines that are still needed, or bring in cache lines that are not needed, leading to cache pollution. Prefetching directly to L1 comes at a substantial area cost because of all the logic required to do the background filling of the cache.
- Block Prefetch: If the data that is required is highly predictable, then Block Prefetch can provide significant performance advantages if the data can be brought into the cache ahead of when it is needed, while working on other tasks in the meantime. Care must be taken to not cause cache thrashing in the cache by bringing in too much data ahead of time. The block Prefetch and Modify (also known as Prepare to Store) operation can save considerable bandwidth and latency for structures that will be completely overwritten, since data does not need to be requested over the bus. The block downgrade operation DHI.B can save cache writeback bandwith by simply dropping data that is no longer needed.

18. Xtensa RAM Interface Ports

The cache interface port described in the preceding chapter provides fast local memory in a manner transparent to the system programmer. However, cache misses trigger PIF accesses, which are much slower in servicing memory requests than local memory ports. Local RAM interface ports provide instructions or data to the Xtensa processor core in the shortest time possible.

Note: The ports described in this chapter may be connected only to blocks with memory semantics. The processor will occasionally issue speculative loads to these ports and actual loads in the instruction stream are occasionally generated multiple times. There are no signals to indicate whether the loads are speculative or real. Stores to the ports described in this chapter are never speculative but may be delayed by several cycles due to write buffers and queues within the processor.

18.1 Local Instruction RAM and Data RAM Ports

A synchronous instruction or data RAM with 1-cycle latency (for a 5-stage pipeline) or 2-cycle latency (for a 7-stage pipeline) is connected to a RAM interface port and is mapped to a contiguous address range whose start address must be an integer multiple of its size. The instructions or data are accessed when the fetch or load/store address calculated by the processor core falls within the configured address range of the RAM. As many as two instruction RAMs and two data RAMs can be instantiated per Xtensa core.

Note: There is a limit of six instruction-memory arrays and six data-memory arrays for any given configuration. Each data-memory array can be configured as a single or multiple (2,4) banks. (Note that cache ways are considered memory arrays in this limit).

There are a few rules governing the placement of these memory arrays in the processor's address space:

- Their address spaces cannot overlap.
- Their start addresses must be an integer multiple of their size.
- Configurations with MMU with TLB have additional requirements as described in Section 3.2 “MMU with Translation Look Aside Buffer Details” on page 19.

- In general, all data memories must be enabled in the E-stage of a load instruction, because the actual target is not known until a full address comparison is performed in the following stage. However, configurations with Region Protection or Region Protection with Translation (Section 3.1) Memory Management can have more accurate data memory power usage by following these rules:
  - For data accesses, each of the eight 512MB regions are allocated to either data cache or local data RAM.
  - Regions allocated for local data RAM are set to the bypass attribute.
- In configurations with the MPU, similar power savings can be achieved by:
  - Again allocating each of the eight 512MB regions to either data cache or local data RAM.
  - Using special register 98, CACHEADRDIS, to disable each of the 512MB segments of memory for data cache access.

By following the latter two rules, the load/store unit will know whether to activate the data cache or data RAMs. In addition, if multiple data RAMs are configured, a quick address check is performed to determine which data RAM is the actual target.

As with the cache interface, a RAM port's address and control signals (and the data lines for a write) are presented during one cycle, and the RAM returns read data either one cycle later (5-stage pipeline) or two cycles later (7-stage pipeline).

Note: For 7-stage pipeline configurations with a 2-cycle memory-read latency, a write operation immediately followed on the next cycle by a read to the same address must still return the newly written data, not the old data.

Because loads from RAM compete with stores to the RAM, stores pass through a store buffer in the processor so that loads and stores can somewhat overlap. The store buffer is a shared queue of write operations to the data-cache, data-RAM, and the XLMI ports. There is always room in the store buffer to hold any stores that might be deferred due to pending loads.

Most stores go through the store buffer, however cache-line castouts and stores to the local instruction-RAM as well as writes to instruction-cache, and instruction-tag memories bypass the store buffer. All stores directed to the processor's PIF also go through the write buffer.

Memory accesses to both the instruction- and data-RAM ports can be blocked by use of the Busy signals: IRam*n*Busy and DRam*n*Busy*m*. (See Section 18.2 for busy-signal naming conventions.) Busy signals indicate that, on the previous cycle, the corresponding instruction or data RAM was unavailable. Such a signal might indicate, for instance, that the RAM was being used by an external entity or, for processor configurations with two load/store units, that the RAM is being accessed by the other load/store unit. Assertion of a Busy signal ends a transaction. Following assertion of a Busy signal to end a

transaction, the processor may elect to retry the transaction during the next cycle; it may retry the transaction later, possibly with other intervening bus transactions, or it may abandon the transaction. In any case, the target hardware need not save the transaction address because the processor will present the target address again when (and if) it retries the transaction.

Note: A Busy signal cannot be used to stretch a memory-access cycle. It terminates the current memory cycle. The processor may choose to rerun the cycle.

This mechanism allows several processors to share one RAM and it permits the use of single-ported instead of dual-ported RAMs as illustrated in Chapter 23, “Local-Memory Usage and Options” on page 425. It might also allow external DMA devices to access the RAM. More details on Busy signal behavior and use appear in the Section 18.9 “Data RAM Transaction Behavior” on page 314, and Section 23.3 “Memory Combinations, Initialization, and Sharing”.

If the PIF option is configured, it is also possible to allow inbound-PIF requests to read from and write to the local instruction and data RAMs. This feature makes it possible to initialize the local RAMs through the processor’s PIF (see Section 32.8 “Global Run/Stall Signal”, which describes the Run/Stall feature).

It is important to note that there is a combinational logic path between all Busy signals to all of the processor’s memory-address and memory-control lines and to the address and control lines of the XLMI port. Consequently, the state of a Busy signal in cycle n is used during that same cycle to determine the state of the address and control signals to all local memories and the XLMI port. This in turn implies that under no circumstances should a Busy signal’s state during any cycle be based on the state of the address and control signals during that same cycle, because such a condition would lead to a combinational-logic-only loop. Ideally, external logic should determine the state of the Busy signal in cycle n+1 from the state of the associated address and control lines during cycle n, and the Busy signal itself should be registered before being fed back to the processor. The system designer must ensure that the implementation and timing of external circuits meet these requirements to ensure the proper synchronous operation of the overall system.

Memories and other devices with read side-effects (such as a FIFO) must not be connected to RAM interface ports because of the speculative nature of the Xtensa processor’s read operations. Xtensa processor loads are speculative because a read operation to RAM does not necessarily mean that the processor will consume the read data and retire the load instruction. For performance reasons, many loads will attempt to read data from the RAM port even though the actual load target address is assigned to another memory interface. A variety of internal events and exceptions can cause pipeline flushes that subsequently cause the processor to replay loads targeted at addresses assigned to the RAM ports because the data obtained from the first load has been flushed. Consequently, all devices attached to the processor’s RAM ports must accommodate the speculative nature of all processor read operations.

18.2 Signal Descriptions

See Table 10–18 and Table 10–21 for the instruction and data RAM interface port signals. The signal-naming convention includes the memory name, the memory number denoted by *n*, and an optional suffix *m* for data memories. This configuration-dependent suffix *m* can represent the load/store unit number or the physical data RAM bank ID. For example, `DRam0Addr1` is the name for the address lines to `DataRAM0` driven by load/store unit 1, while `DRam0AddrB1` is the name for the address lines to bank 1 of `DataRam0`. As many as two optional instruction RAMs and as many as two optional data RAM ports can be configured. Some of the RAM signals are not present unless other configuration options are also selected in addition to the instruction- and data-RAM options. These requirements are described in the Notes columns of Table 10–17 and Table 10–20.

Note:

- See Section 35.7 on page 630 for information about RAM interface AC timing.
- For configurations with multiple load/store units, the data RAM interface can be opened to permit you to design your own C-Box. For such a case, data RAM read and write interfaces are split. Refer to Section 18.10 “Exposed Processor Interface for a Customer-Designed C-Box” for information.

18.3 Instruction RAM Data-Transfer Transaction Behavior

The processor’s instruction RAM array timing is the same as for the cache data and tag memory arrays. Figure 18–69 shows the relationship between the instruction RAM port’s control and address lines, and the data returned from the instruction RAM. The processor drives the address and control signals during one cycle, and the instruction RAM returns the data during the next cycle. For a write operation, the processor presents the write data to the RAM along with the address and control signals. Note that with store instructions, the instruction RAM is always written 32-bits at a time.

Inbound PIF writes are 32-bits at a time when the instruction RAM is 32-bits wide, either 32 or 64 bits at a time when the instruction RAM is 64-bits wide, and are 32-, 64-, or 128-bits wide when the instruction RAM is 128 bits wide. In the case of instruction RAMs that are 64 or 128 bits wide, there is also word-enable control, with one bit of control for every 32-bits of data. The word enables are also active during instruction RAM reads. Therefore, these word enables can be used to directly drive the memory-block enables when 64- or 128-bit memories are constructed using two or four 32-bit RAM arrays.

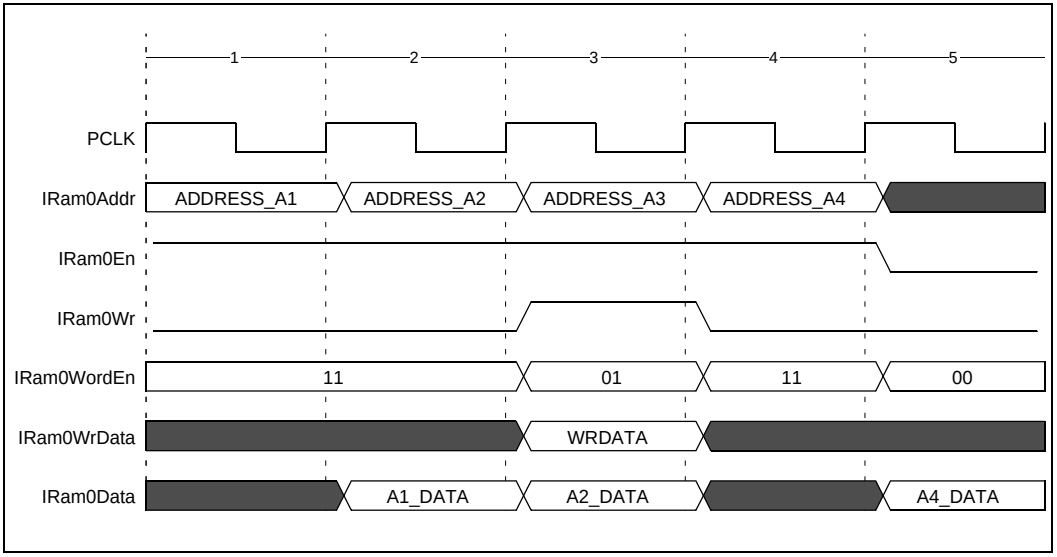


Figure 18–69. Instruction RAM Access (64-bit Wide RAM Arrays, 5-Stage Pipeline)

Figure 18–70 shows the same instruction-access operation but for a 7-stage pipeline, which allows one extra cycle of delay for data to return from memory during a memory read. Note that the implementation of 2-cycle memory must be such that a write followed immediately by a read to the same address must return the newly written data, not the old data.

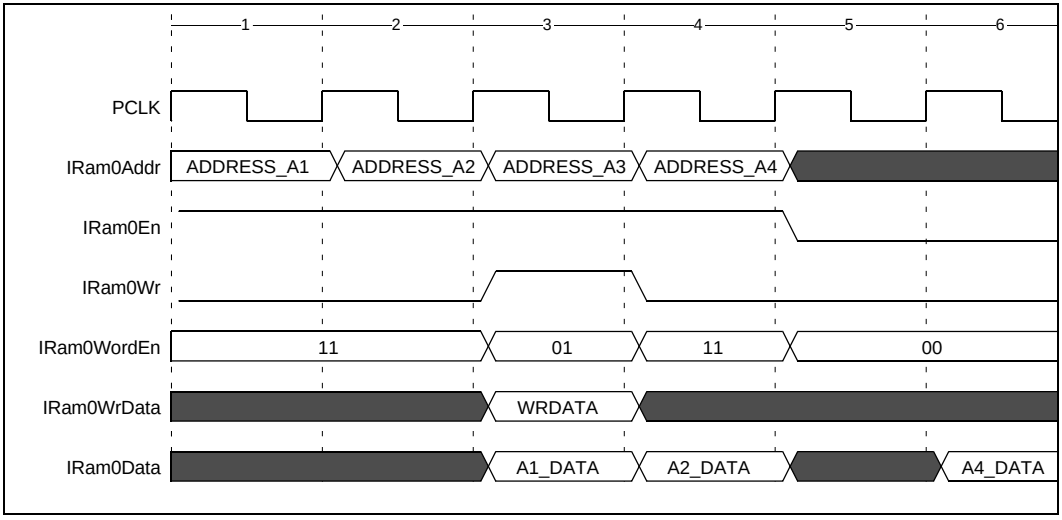


Figure 18–70. Instruction RAM Access (64-bit Wide RAM Arrays, 7-Stage Pipeline)

18.3.1 Instruction RAM Load and Store

Instruction RAM and ROM can be referenced as data by the L32I, L32R, and S32I instructions. However, except for certain cases of L32R (as discussed in the following paragraph), the access latency of these instruction memories is much slower than that of data cache or the local data memories, because data-load operations from and store operations to instruction RAM are implemented by replaying the load or store instruction. For this reason, using local instruction memories for data storage is not a recommended practice. However, this feature is useful for initially loading or subsequently changing instruction memory, as required by the application.

When an instruction RAM approaches or exceeds 256 KBytes, accessing literals in data memory with L32R becomes more difficult due to the limited range of this instruction. In this case it is desirable to place the literals in the instruction RAM with the rest of the code, but still not suffer the penalty of a replay when reading a literal with L32R. For this reason in most cases L32R to an instruction RAM will execute without a replay, but may suffer a single cycle stall in the case that instructions also need to be fetched on the same cycle as the load from instruction RAM. Whether an L32R instruction will execute without a replay depends on the memory protection and translation option chosen, as well as the instruction RAM memory error option:

- Region protection: L32R instruction RAM will always execute without a replay, except in the case that the memory error configuration option is selected, and a memory error is detected.
- Region protection with Translation: L32R to instruction RAM will execute without a replay if the load address it is accessing is in the same 512MByte region as the instruction itself. If the L32R is accessing an instruction RAM or ROM that is in some other 512MByte region, then it will replay. Note that this would not be a common case, as literals are in general in the same instruction RAM as the code. A replay will also occur in the case that the memory error configuration option is selected, and a memory error is detected.
- MMU with Translation Look Aside Buffer (TLB) and Autorefill: L32R to instruction RAM always replays. Note that the instruction RAM size is limited to 128KBytes of this memory protection and translation option is chosen, thus literal placement is not generally a problem.

Note:

- Unaligned load/store accesses to instruction memory will cause the processor to take an exception.
- Load/store accesses to instruction RAMs and ROM are somewhat restricted with respect to bundled FLIX instructions. They must be in slot 0, and in the case of two load/store units, they cannot be bundled with another load or store, otherwise a load/store exception will occur.

Figure 18–71 shows a typical pipeline diagram for executing the `L32I` load instruction with an instruction-RAM target address. The load latency is 6 cycles (7 cycles if the instruction is unaligned) instead of the usual 1-cycle latency. During cycle 3, the processor computes the load's target address. This target address is output only to the local data-memory ports (data cache, data RAM, data ROM, and XLMI) during this cycle. The term <DataMem> in Figure 18–71 applies to all of the configured local data memories.

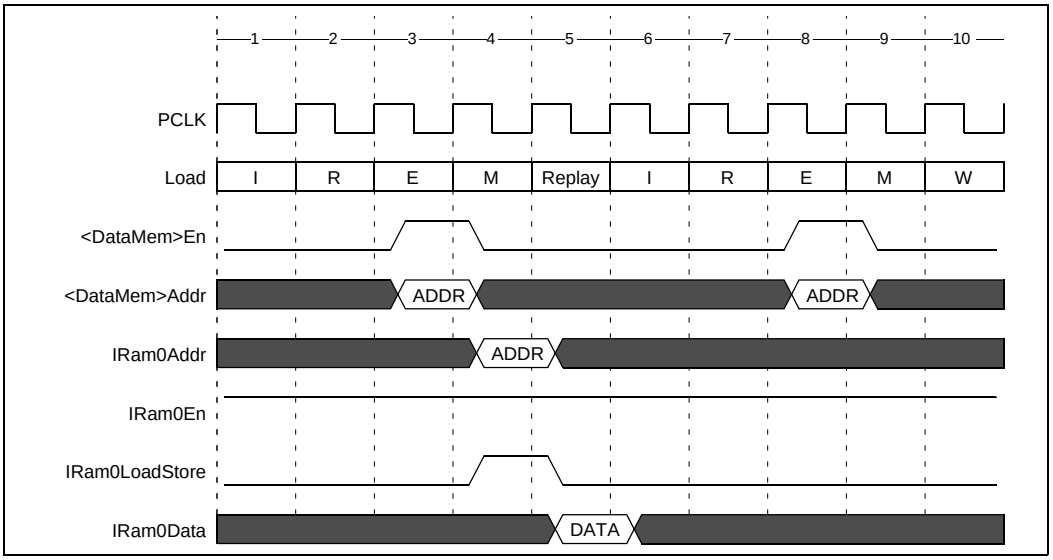


Figure 18–71. Instruction RAM Load with Replay (5-stage Pipeline)

During cycle 4, the processor compares the target load address to the instruction RAM region to see if the instruction accesses instruction RAM. If the TLB is configured, the target address is translated (memories are physically tagged) before the address is compared. After it's determined, in cycle 4, that the load address is within the instruction-RAM region, the address is output to the instruction memory on `IRam`*n*`Addr`.

During cycle 5, the instruction RAM returns the data. However, it's too late to commit this data to the processor's AR registers, so the data is stored in a temporary register and the instruction is replayed. Cycles 6 through 10 are replayed cycles for the load instruction. During cycle 9, the data saved previously in a temporary register is still available, so a second instruction-RAM access cycle is not needed or generated. During cycle 10, the data held in the temporary register is written to the processor's AR register file and the load instruction completes.

The `IRam0LoadStore` signal differentiates between an instruction doing a data load or store to instruction RAM, and the instruction fetch logic using the instruction RAM to fetch instructions.This signal can be used by an external agent to give priority access to the instruction RAM to the local processor (in the case that the instruction RAM is being shared between several processors for instance). This signal can remain unconnected if

there is no arbitration for the instruction RAM, or if the arbitration chooses to not use this information in its decision. In the example `IRam0LoadStore` is high in cycle 4, indicating that the processor is attempting to load from the instruction RAM. This signal does not go high in cycle 9 because the instruction RAM is not being accessed and the captured data is being used. In all other cycles except cycle 4 the instruction RAM interface is being used by the instruction fetch portion of the pipeline.

For a 7-stage pipeline, the access behavior is the same as for a 5-stage pipeline with the Load address and control going out in the M-stage of the pipeline, but the instruction memory returns the data two cycles after the address is presented rather than one cycle.

Figure 18–72 shows a typical pipeline diagram for executing the L32R load instruction with an instruction RAM target address and no replay. Instructions are being fetched from the instruction RAM (I0ADDR, I1ADDR, I2ADDR), when a L32R instruction comes down the pipeline. When the load is in the R-stage, the address is precalculated and it is determined whether the address is going to the instruction RAM. For sake of illustration, a stall condition is shown where the instruction fetch pipeline also requires access to the instruction RAM when the load instruction is in the E-stage in this figure. In cycle 3 the address goes out for the L32R instruction, and in cycle 4 the address goes out for the instruction fetch. The instruction then completes in the normal way.

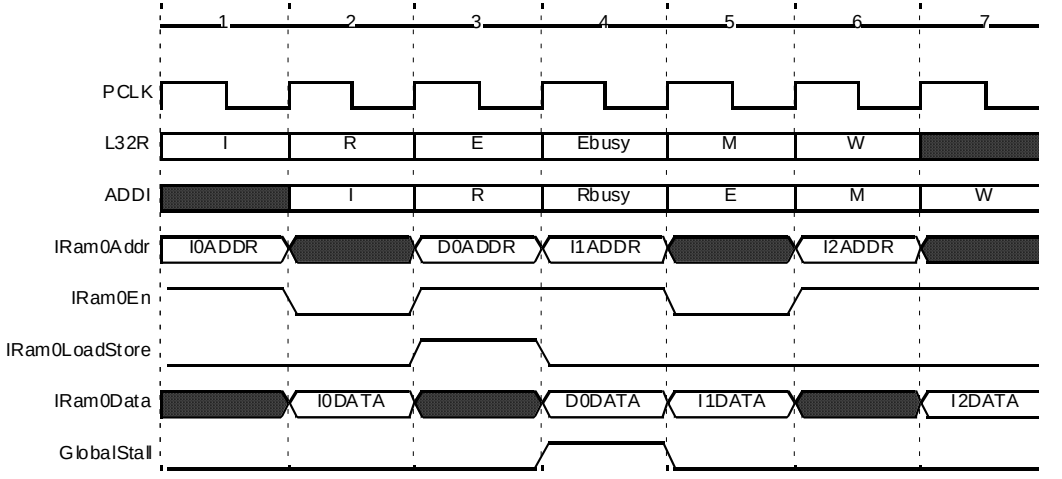


Figure 18–72. Instruction RAM L32R Load without Replay (5-stage Pipeline)

As in the previous diagram, the `IRam0LoadStore` signal differentiates between an instruction doing a data load or store to instruction RAM, and the instruction fetch logic using the instruction RAM to fetch instructions.

For a 7-stage pipeline, the access behavior is the same as for a 5-stage pipeline with data coming back two cycles after the address goes out.

Figure 18–73 shows the diagram for executing an S32I store operation on an instruction-RAM address. The store latency is effectively 6 cycles (7 cycles if the instruction is unaligned), because the processor replays the instruction following the store. During cycle 3, the processor computes the target address. During cycle 4, the processor compares the target address to the instruction-RAM region to see if the operation accesses instruction RAM. If the TLB is configured, the target address is translated (memories are physically tagged) before the address is compared.

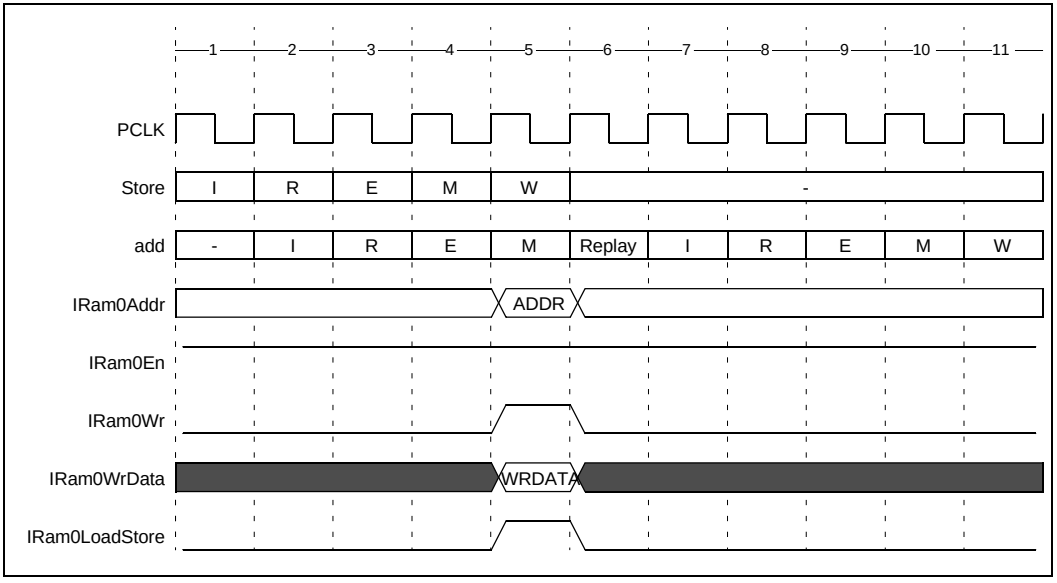


Figure 18–73. Instruction RAM Store (5-Stage Pipeline)

During cycle 5 (the W stage of the store instruction, when the processor has determined that the instruction-RAM store can proceed), the processor presents the address and control signals and the write data to the instruction RAM. However, this store operation conflicts with an instruction fetch, also scheduled for that same cycle by the instruction-fetch unit. The processor resolves this conflict by flushing its pipeline and replaying the instruction immediately following the instruction-RAM store. The replay starts in cycle 6. During all other cycles except cycle 5, the instruction RAM interface port is used for instruction fetches.

Note that the processor asserts IRam0LoadStore during cycle 5, indicating that it is attempting to perform a store operation to the instruction RAM. An external agent can use this signal to give priority access to the processor (in the case that the instruction RAM is being shared between several processors, for instance).

The behavior is the same for a 7-stage pipeline as for a 5-stage pipeline with the store address, control, and data going out during the pipeline's W stage. The effective instruction latency is 7 cycles, or 8 cycles if the instruction is unaligned.

(**Note**: In a multi-slot FLIX instruction, instruction-RAM and -ROM transactions can only be initiated by a load or store operation executed from slot 0 in the FLIX instruction bundle. Otherwise, a load/store exception will occur.)

18.3.2 Instruction Memory Busy Functionality

The instruction RAM's IRam*n*Busy and the instruction ROM's IRom0Busy signals are inputs to the processor core that allow external agents such as DMA controllers or other processors to share the local instruction memories. For example, busy functionality can be used in multi-bank and multi-agent systems where there is a requirement to share an instruction RAM among multiple Xtensa processor cores. External agents can access the instruction RAM at any time and for any time duration. These external agents simply use the busy signal to indicate that the instruction RAM was in use the previous cycle. Busy functionality must be explicitly selected as a configuration option and can incur a cycle-time penalty. This section only describes IRam*n*Busy. Other than the store transaction, IRom0Busy functions identically to IRam*n*Busy.

Use of IRam*n*Busy assumes the presence of an external agent that arbitrates instruction RAM use and manages the external multiplexing of the address, control, and data signals going to the instruction RAM. This external logic can affect the address and data timing paths, so careful timing analysis is required during design. See Section 23.3 "Memory Combinations, Initialization, and Sharing" for examples of how to use IRam*n*-Busy.

If the external agent grants control of the instruction RAM to something other than the Xtensa processor, it must assert IRam*n*Busy during the next cycle to signal the processor that the instruction RAM was not available during the previous cycle. If the processor needed data from the instruction RAM during that cycle, it must re-request it. Figure 18–74 shows this process, where the external agent that controls access to the instruction RAM asserts IRam*n*Busy one cycle after the processor drives the instruction RAM's address and control lines for the corresponding instruction fetch. For the case shown in Figure 18–74, the processor needed data from the instruction RAM, so it maintains the state of the address and control signals until it gains access to the instruction RAM. Note that IRamBusy is always asserted one cycle after the RAM is used by an external entity.

It is important to note that there is a combinational path between the IRam*n*Busy input and the instruction RAM address and control output signals that maintains the state of these outputs at their previous values if necessary. Consequently, logic external to the processor that generates IRam*n*Busy cannot depend on the immediate state of the processor's instruction-RAM memory-address and -control outputs during the same cycle

that IRam*n*Busy is asserted. The state of IRam*n*Busy must depend only on the state of these address and control outputs during the previous cycle. In other words, external logic that generates IRam*n*Busy must derive the proper state of IRam*n*Busy using the state of the memory-address and memory-control outputs during the previous cycle. The IRam*n*Busy signal should be registered before it is supplied back to the Xtensa processor. The system designer must ensure that the implementation and timing of external circuits meet these requirements for the proper synchronous operation of the overall system.

Note that assertion of IRam*n*Busy ends the transaction. Following the assertion of IRam*n*Busy and subsequent termination of the transaction, the processor may elect to retry the transaction during the next cycle; it may retry the transaction later, possibly with other intervening bus transactions; or it may abandon the transaction. In any case, the target hardware need not save the transaction address, because the processor will present the target address again when (and if) it retries the transaction. The processor can change the IRam*n*Addr output values after the termination of the transaction by IRam*n*Busy, because the processor pipeline does not necessarily stall when IRam*n*Busy is asserted. Pipeline events such as taken branches, exceptions, and pipeline replays may cause the IRam*n*Addr signals to change after a transaction is terminated.

18.3.3 IRamnBusy Operation During an Instruction Fetch

IRam*n*Busy causes the processor to stall only when it attempts to access instruction RAM either for an instruction fetch or because it is executing a load or store instruction that can access instruction RAM (L32R, L32I, L32I_N, S32I, and S32I_N). Thus, if the instruction-fetch address does not correspond to an instruction RAM address (the processor may be fetching instructions from another local memory such as an instruction cache), then assertion of IRam*n*Busy has no effect on the processor. Even during an instruction fetch, the processor may have enough bytes stored in its internal instruction buffers to avoid stalling when accessing instruction RAM. In general, when the processor is executing instructions out of instruction RAM, it requires access to the instruction RAM most of the time and the conditions under which it will avoid a stall are limited.

Note that if instruction RAM access is needed, the state of the address and control signals will be maintained or repeated during the same cycle that IRam*n*Busy is asserted. This temporal coincidence stems from a combinational path in the Xtensa processor between the IRam*n*Busy signal and the instruction RAM's address and control signals.

Figure 18–74 shows an example where IRam*n*Busy does not stall the pipeline. A jump instruction executed and changed the address being used to access the instruction RAM.

The following conditions can cause the address to change during IRam*n*Busy:

- `Taken Branch`, `Jump`, `Call`, `Return`, or `Loopback instruction`
- `Exception`, `Interrupt`, or `Replay`

For these cases, the fetch unit drives the new target address.

Note: When the OCD (on-chip debug) mode is active, `IRam`*n*`Busy` does not stall the pipeline because the instruction fetch does not access instruction RAM.

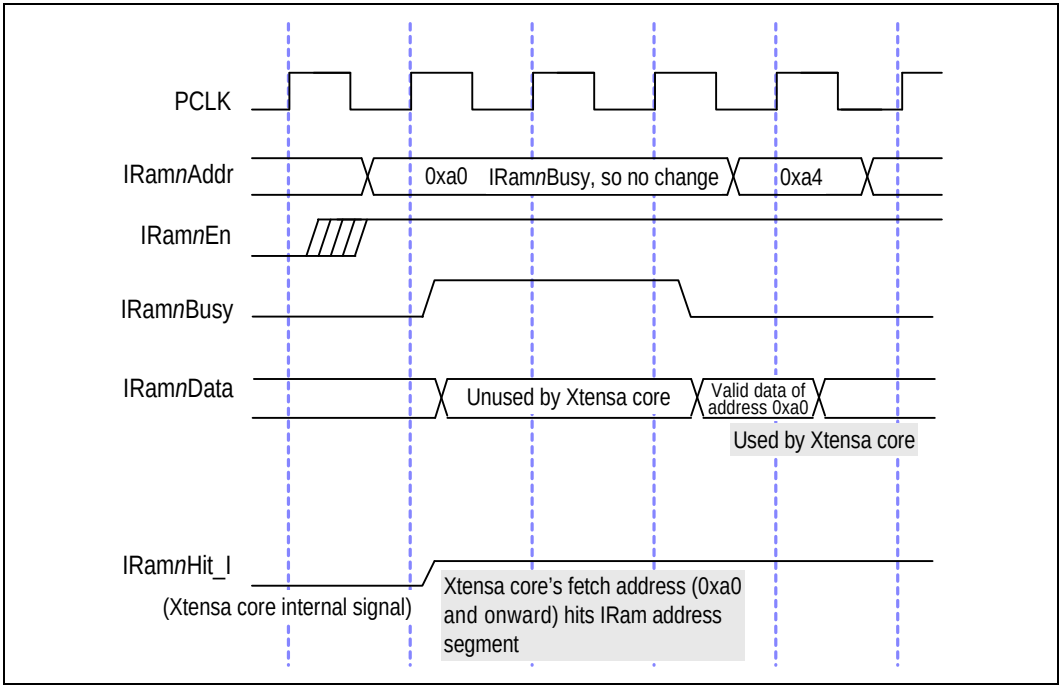


Figure 18–74. Interaction of an Instruction RAM Access and IRam*n*Busy

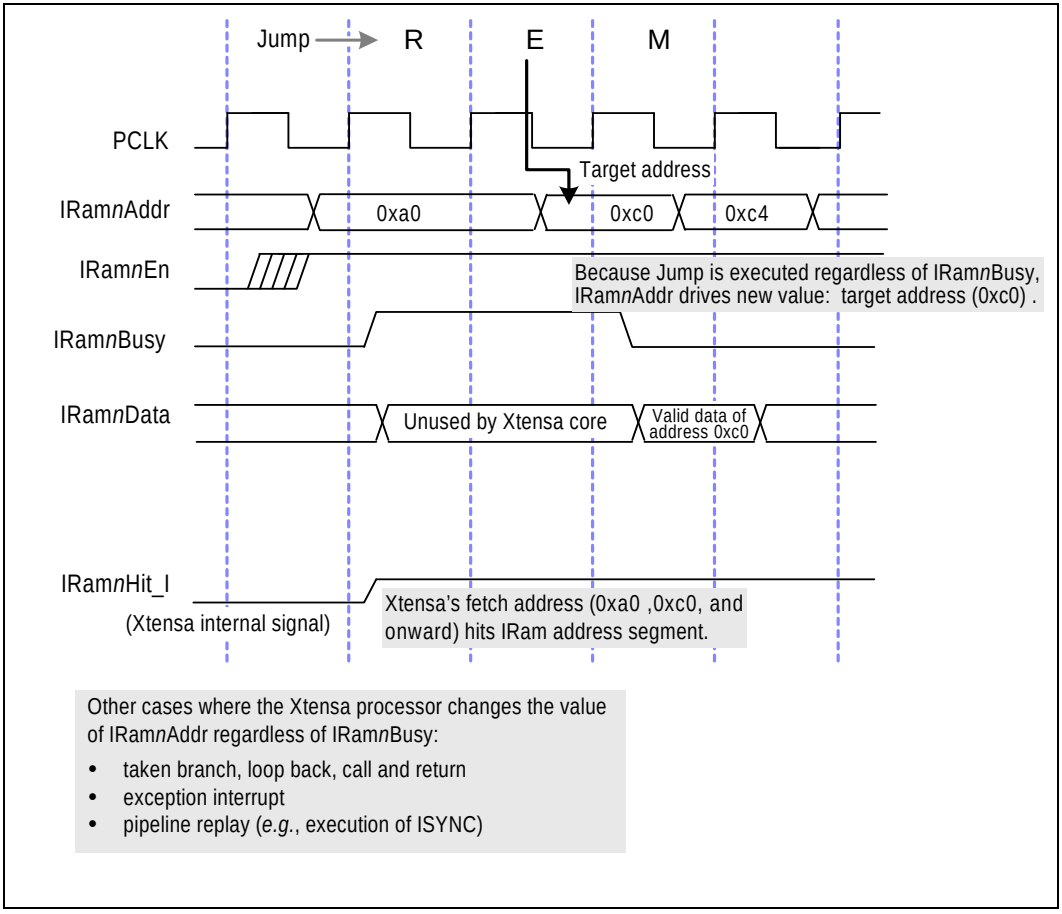


Figure 18–75. IRam*n*Addr Can Change While IRam*n*Busy is Active

18.3.4 IRamnBusy Operation During Loads and Stores

As Figure 18–76 shows, an instruction-RAM-load instruction will stall the pipeline just like an instruction fetch from the instruction RAM. An instruction-RAM-store instruction will repeatedly attempt to store data to the instruction RAM as long as `IRamnBusy` is asserted, as shown in Figure 18–77.

Note: This section's pipeline diagrams use smaller letters to denote stages where pipeline stalls are occurring.

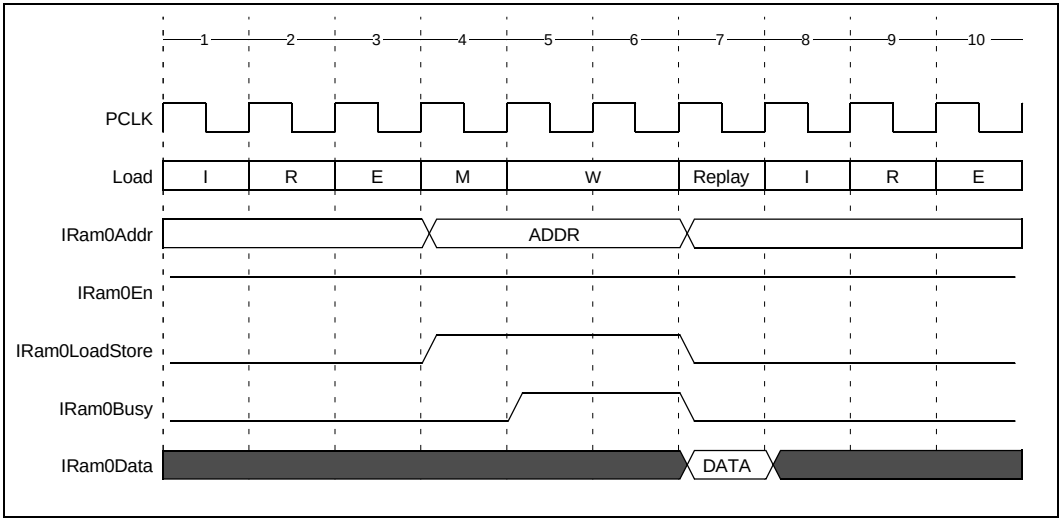


Figure 18–76. Instruction RAM Load with IRam*n*Busy (L32I with Replay Case)

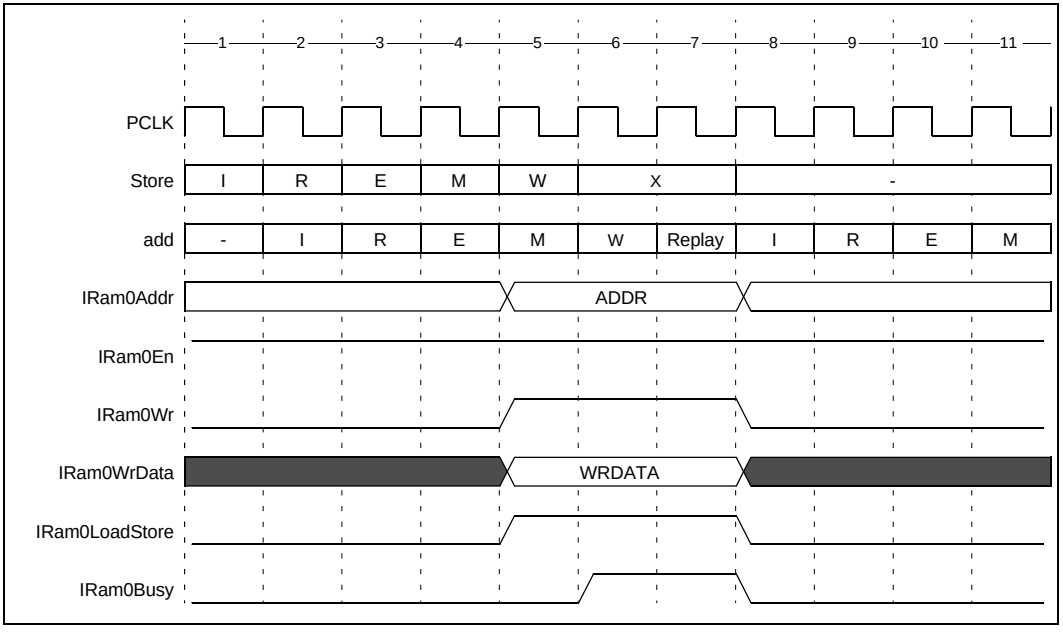


Figure 18–77. Instruction RAM Store with IRam*n*Busy

18.3.5 Inbound PIF Access to Instruction-RAM

Inbound PIF logic can read and write the instruction RAM in response to inbound PIF requests. When this occurs, the instruction fetch logic and the pipeline load/store logic effectively receives an internally generated `IRam`*n*`Busy` signal, and the instruction RAM interface is taken over by the inbound PIF logic to do the read or write. Thus the processor may or may not stall during an inbound PIF to an instruction RAM depending on whether it needs access to that instruction RAM or not.

The inbound PIF to instruction RAM always has higher priority than instruction fetch or an instruction load/store to instruction RAM. If an external `IRam`*n*`Busy` occurs, then the inbound PIF access to the instruction RAM will be retried until the instruction RAM becomes free.

Note that write cycles coming from the inbound-PIF logic to an instruction RAM can perform an aligned 32-bit word write (at minimum), but can also perform writes of aligned 64-bit quantities if the PIF width is 64-bits and the instruction RAM width is 64 or 128 bits, or of aligned 128-bit quantities if the PIF width is 128 bits and instruction RAM width is 128 bits. Inbound PIF is a fast and convenient way to initialize instruction RAMs.

18.4 Instruction RAM Memory Integrity

The instruction RAM can optionally be configured with memory parity or error-correcting code (ECC) for widths up to 128 bits. These options add extra memory bits that are read from and written to the RAM. The processor uses the extra bits to detect errors that may occur in the cache data (and sometimes, to correct these errors as well). The parity option allows the processor to detect single-bit errors. The ECC option allows the processor to detect and correct single-bit errors and to detect double-bit errors.

If a memory error is detected on an instruction fetch, either correctable or uncorrectable, an exception is signalled when instruction uses a byte from that instruction fetch. Effectively, the instExc bit of the MESR register is tied to 1.

If a memory error is detected on a load from IRAM, there are two possible situations:

- Uncorrectable error detected or correctable error but the `DataExc` bit of the `MESR` register is set: on the replay of the instruction, a memory error exception will be signaled in the W-stage.
- Correctable error detected and `DataExc` bit of the `MESR` register is not set: the load from IRAM will replay twice rather than once, logging information about the memory error on the first replay, and then correcting the memory error and retiring normally on the second replay.

Note: Although the ECC logic will correct the data in this case, the erroneous value will still reside in memory. It will be corrected again if the processor reads that location again but to correct the value in memory, the processor must explicitly write the corrected value to memory.

If a memory error is detected on a load from IRAM coming from inbound DMA, there are two possible situations:

- When an uncorrectable error is detected, the PIF will return a bus error to the external master.
- When a correctable error is detected, it will be corrected on the fly, but the erroneous value will still reside in memory.

In both cases, the memory error information is not logged internally in the processor.

18.5 Different Connection Options for Local Memories

A Connection Box (C-Box) option is available for configurations that have two load/store units. If the C-Box option is selected, the Xtensa processor exploits sophisticated data routing and arbitration mechanism in the C-Box to allow a multiple-load/store-unit processor to directly connect with single-port data RAMs and/or data ROM. Figure 18–78 illustrates how the C-Box accepts requests from different agents and converts them for single-ported data RAM interface. The C-Box also supports the data RAM/ROM banking feature (Section 18.6) by allowing multiple data RAM or data ROM banks (up to four) to be directly connected to the Xtensa processor. In general, the C-Box provides a convenient way to connect multiple agents with single or multiple-bank data RAM and data ROM by providing built-in arbitration and data multiplexing mechanisms.

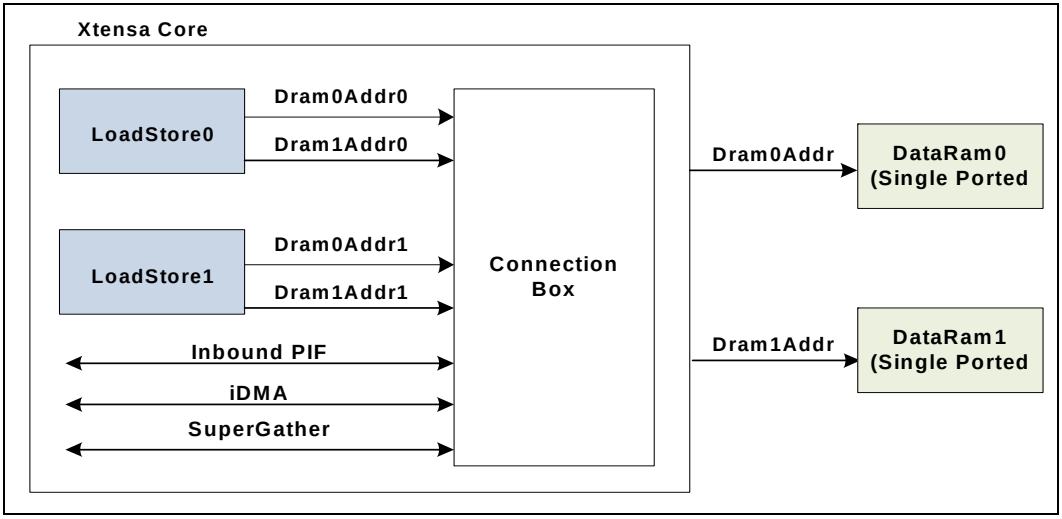


Figure 18–78. Connect Multiple Agents to Data RAM with the C-Box

The C-Box detects access conflicts and arbitrates among the following agents:

- Load/store units
- Inbound PIF
- iDMA
- SuperGather

Upon detection of potential access conflicts among simultaneous requests, the C-Box arbitrates the requests per bank and grants accesses to the data RAM (data ROM) banks to the successful request. It also sends a busy signal to the unsuccessful request in the next cycle. In a single arbitration cycle, there can be multiple requests granted access to different banks simultaneously. C-Box arbitrates the conflicts based on the following priority preference (from high to low):

- High-priority inbound PIF/iDMA
- Load instructions
- SuperGather
- Store instructions
- Low-priority inbound PIF/iDMA

Note that such an arbitration preference can be overridden by dynamic factors. For example, when an memory operation needs to be performed atomically with regard to other operations.

In general, load operations always have higher priority over store operations. This minimizes the chance that the Xtensa pipeline will be stalled waiting for data. When two load/store units issue the same type of transaction (both loads or both stores) to the same bank, the result is determined as follows:

- If they are both loads, load/store unit 0 gets precedence
- If they are both stores, which ever has more store buffers occupied gets precedence. Note that, for timing reason, the number of occupied store buffers is registered in flip-flops before being used for comparison.

Data ROM only supports core loads. Load conflicts are resolved by giving load/store unit 0 precedence.

Note: To maximize processor throughput, the Xtensa core can hold data in the store buffer, thus giving priority to loads. The Xtensa processor can bypass the data in the store buffer if a younger load has dependence on it. The data in the store buffer will be dispatched to data RAM at the earliest opportunity. Therefore, there is an indeterministic timing window between when the store instruction is committed and when the data is dispatched to data RAM. To avoid this, MEMW can be used to flush a store buffer, but this is at a performance cost.

If an inbound-PIF transaction does not have an access conflict with other agents, it is allowed to proceed independently. If there is a conflict with the pipeline, its priority is determined by a priority field PIReqPriority[1:0]. 2’b11 gives the inbound-PIF transaction the high priority. Low priority inbound PIF transactions (2’b00, 2’b01, 2’b10) will eventually graduate to high priority if repeatedly denied access. Different bandwidths are allocated to these low priority inbound PIF transactions based on PIReqPriority[1:0]. For more details of inbound PIF bandwidth relative to core loads and stores, see Table 18–66.Use high-priority inbound PIF (PIReqPriority=2’b11) accesses with caution, as issuing a long sequence of back-to-back high-priority inbound PIF accesses may stall the Xtensa processor if they both compete for the same data RAM bank.

S32C1I is treated as a load instruction during arbitration. The priority of an inbound-PIF RCW transaction is determined by PIReqPriority[1:0]. When they both try to lock a data RAM bank at the same time, they are arbitrated according to the same priority order as listed above. However, if they are locking different data RAM banks, they are allowed to proceed independently.

iDMA’s data RAM accesses are low-priority. If the iDMA does not have an access conflict with other agents, it is allowed to proceed independently. In cases where the iDMA’s access is continuously denied, iDMA can elevate to high priority, thus allowing it to be granted quickly.The agent not allowed access (informed by the Busy mechanism described in Section 18.9.2) can retry for access.

For two load/store configures, there is also an option if you want to design your own C-Box externally to the Xtensa core. This is achieved by setting C-Box option to *false*. Doing so will export a group of internal load/store interface signals to be used by an external arbiter. Please refer to Section 18.10 “Exposed Processor Interface for a Customer-Designed C-Box” for details.

18.6 Banking for Local Memories

Partitioning a monolithic local memory into multiple physical banks can potentially improve memory access throughput. Multiple-banks allow load/store units and inbound PIF to access data RAM in parallel as long as these requests target to different physical banks. Therefore, the banking feature can potentially improve the overall local memory bandwidth significantly.

Figure 18–79 illustrates a mode where two load/store units and the inbound PIF all request to access data RAM at the same cycle. Given that these requests target different banks, a 4-bank data RAM can service these requests simultaneously, thus maximizing the memory access throughput.

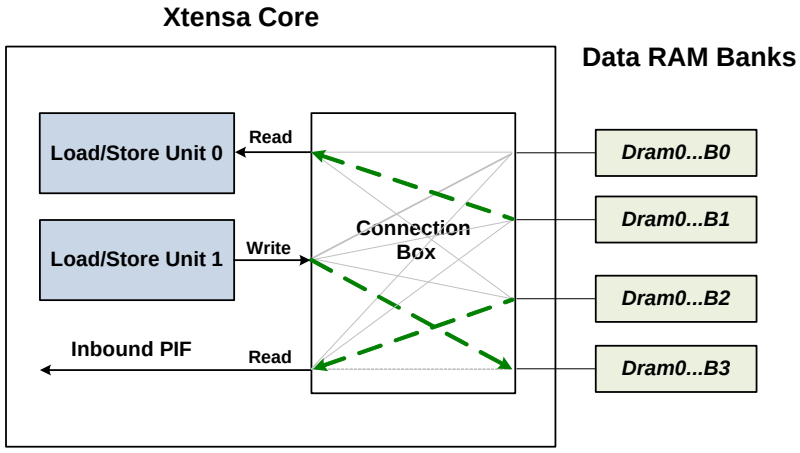


Figure 18–79. Using Multiple Banks to Improve Data RAM Access Throughput

The Xtensa core provides a *banks* option to allow Data Rams/Roms to be banked up to 1,2 or 4 ways. When multiple-banks are configured, arbitration logic and data-path multiplexers are automatically generated. This allows data RAM/ROM banks to be connected directly to the Xtensa core using the standard single-ported data RAM/ROM interface.

For a multiple-bank configuration, the physical data RAM/ROM banks are selected by the address bits just above the dram access `offset` bits, where `offset = log2(dram access width in bytes)`. For example, if the data RAM access width is 8 bytes, then the `offset` is `3`. Specifically, the bit in `ADDR[offset]` is used to distinguish bank B0 and B1 for a two-bank data RAM/ROM. The two bits in `ADDR[offset+1:offset]` are used to distinguish bank B0, B1, B2, B3 for a four-bank data RAM/ROM.

Note: The Xtensa core handles the bank selection internally, thus the exported address port `DRam`*n*`Addr`*m*`/DRom`*n*`Addr`*m* does not contain bank selection bits.

All banks are symmetric; requests from load/store units or inbound-PIF to each bank are arbitrated independently using the rules described in Section 18.5.

Note: When multiple banks are configured, the C-Box is instantiated automatically.

18.7 SuperGather Sub-banks

When the SuperGather engine is configured, a data RAM bank is further partitioned into multiple sub-banks to provide the extra flexibility that allows the SuperGather engine to access each sub-bank with different addresses concurrently.

When data RAM is instantiated as sub-banks, the data RAM arbitration still applies at the bank level. The data RAM sub-banks are invisible and have no effect on accesses from load/store, inbound PIF, or iDMA. However, it enables the SuperGather engine to optimize performance utilizing the extra flexibility provided by sub-banks by accessing sub-banks with different sub-bank addresses.

The physical parameters of a data RAM sub-bank are determined jointly by the data RAM parameters and the SuperGather element width. Their relationship is summarized in Table 10–22, Data RAM Sub-bank Interface Port Signals.

18.8 Load/Store Sharing

When two load/store units are configured, load requests from different load/store units may be merged into a single data RAM read operation. An example of data sharing for a load is shown in Figure 18–80, where two load/store units load from the same address. These two requests are combined into a single data RAM read. The data read from DRam0 Bank1 is shared between these two load/store units. Load/store sharing can improve load/store throughput for certain applications.

As with loads, stores from different load/store units can also be combined into a single data RAM write operation.

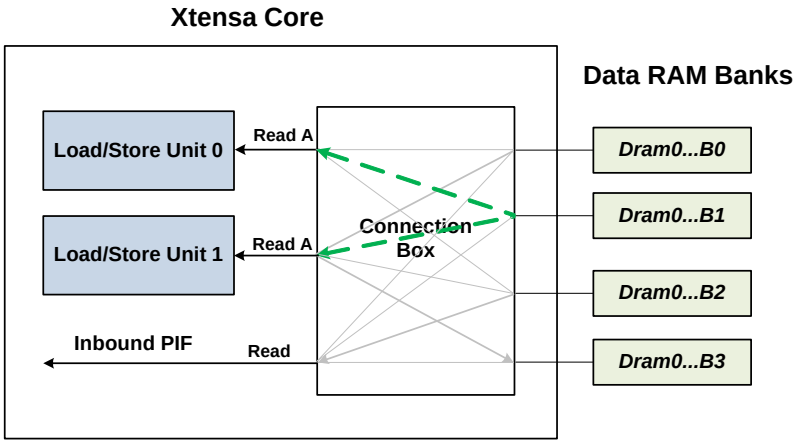


Figure 18–80. Load Sharing Between Two Load/Store Units for Data RAM Access

Note:

- The instruction S32C1I accesses memories atomically. It does not share load or store with other load/store instructions.
- As inbound PIF requests are unrelated to Xtensa pipeline operations, the data sharing feature is restricted only to load/store units. As a result, inbound PIF requests will not share loads or stores with load/store units.

Data sharing is a standard feature for two-load/store-unit configurations with C-Box. To benefit from the data sharing feature, a configuration must have two load/store units and the access address pattern must follow these rules:

- Load sharing: The Xtensa core can combine two loads into a single transaction to data RAM. two loads from different load/store units must target the same data RAM bank and their addresses must match all bits except the bits in the range of `[dramaccessoffset-1:0]`. Refer to Table 18–67 on page 333 for the meaning of dramaccessoffset.
- Store sharing: The Xtensa core can combines two stores into a single transaction to data RAM. There two stores from different load/store units must target the same data RAM bank and their addresses must match all bits except the bits in the range of `[dramaccessoffset-1:0]`.

Note: The compiler can generate FLIX packets suitable for load/store sharing. Xtensa core takes an exception for stores with overlapping byte enables in the same FLIX packet.

To exploit the performance benefit of store sharing, carefully design the store address pattern in FLIX packets so that it does not contain overlapping byte enables.

18.9 Data RAM Transaction Behavior

The basic transactions common to all local Xtensa processor memories are described in Appendix B “Notes on Connecting Memory to the Xtensa Core”. It may be helpful to view that section before reviewing the data-RAM transactions discussed in this section, because the memory transactions are similar and simpler.

18.9.1 General Notes about Data RAM Transactions

To improve pipeline performance, the Xtensa core allows the content of an older store to be bypassed to a younger dependent load. This is called store-to-load bypass. Store-to-load bypass can happen for data-RAM addresses, which causes (with respect to program order) writes to be moved later than reads - while of course maintaining data coherency. To avoid this reordering, the MEMW instruction can be used to separate the load and store instructions, but note that this will have an effect on performance.

A store-to-load bypass also allows a store to be held in the store buffer for multiple cycles before being dispatched to data RAM. To avoid data being held in store buffers, the MEMW instruction can be used to flush the store buffers, however this may also have an effect on performance.

18.9.2 Data RAM Busy

Data RAMs that require data-flow control can use the data-RAM Busy signal. This signal is available only if the Busy configuration option is chosen for the data-RAM port.

Note: When 32-bit ECC/Parity is configured on the data RAMs, the Xtensa processor will perform read-modify-writes on the data RAMs. It is not possible for external logic interfacing with the data RAMs to distinguish these read-modify-writes from normal read/write operations. Therefore, data RAM busy must only be used by the external logic when it accesses a data section that is not being actively worked on by the Xtensa processor or Inbound PIF. Data corruption may result if the external logic intervenes with a read-modify-write sequence.

As with all the other local memory port signals, there is one data-RAM Busy signal associated with each data-RAM bank if multiple banks are configured or with each load/store element. Assertion of this signal when a load is attempted from the data-RAM port will usually stall the pipeline. Assertion of this signal when a store is attempted to the data-RAM port may cause a pipeline stall depending on the state of the associated load/store unit.

The external data-RAM control logic asserts its Busy interface signal one cycle after the processor drives the address and enable lines, regardless of the statically configured memory latency (one or two cycles).

Note: Assertion of the Busy signal in cycle n always applies to the data-RAM transaction begun in cycle n-1. Data-RAM control must base assertion of Busy in cycle n on the registered state of the data-RAM's address and control ports in cycle n-1. The Xtensa processor uses the value of Busy in cycle n to decide what values to place on the data-RAM port's address and control outputs *during cycle n*. In other words, there exist combinational-logic paths within the data-RAM port's logic from its Busy input to *all* of the data-RAM port's outputs including En, Addr, ByteEn, Wr, WrData. Under no circumstances should external logic connected to the processor's data-RAM port assert Busy in cycle n based on the state of the data-RAM's address or control outputs during cycle n, because doing so leads to a combinational-only logic loop. The system designer must ensure that implementation and timing of external circuits attached to the processor's data-RAM port meet these requirements to ensure proper synchronous behavior.

When Busy is asserted, the processor may choose to retry the load or store, or it may choose to try a new load or store. In other words, Busy is a transaction *terminate* signal, and not a transaction extension signal.

Figure 18–81 shows a Busy assertion for a data-RAM load. If Busy is asserted in cycle 4, the processor would probably choose to retry the load. If it so chooses, the output-signal states required for performing the load are repeated, precluding the need for any signal latching by external logic. The processor determines in cycle 4 whether to retry the transaction or proceed to another transaction based on the value of DRam0Busy0 *in cycle 4*. In other words, there is a combinational-logic-only path in Xtensa processors between the DRam0Busy0 signal and the data-RAM port's address and control signals to repeat these output states when the Busy signal is asserted.

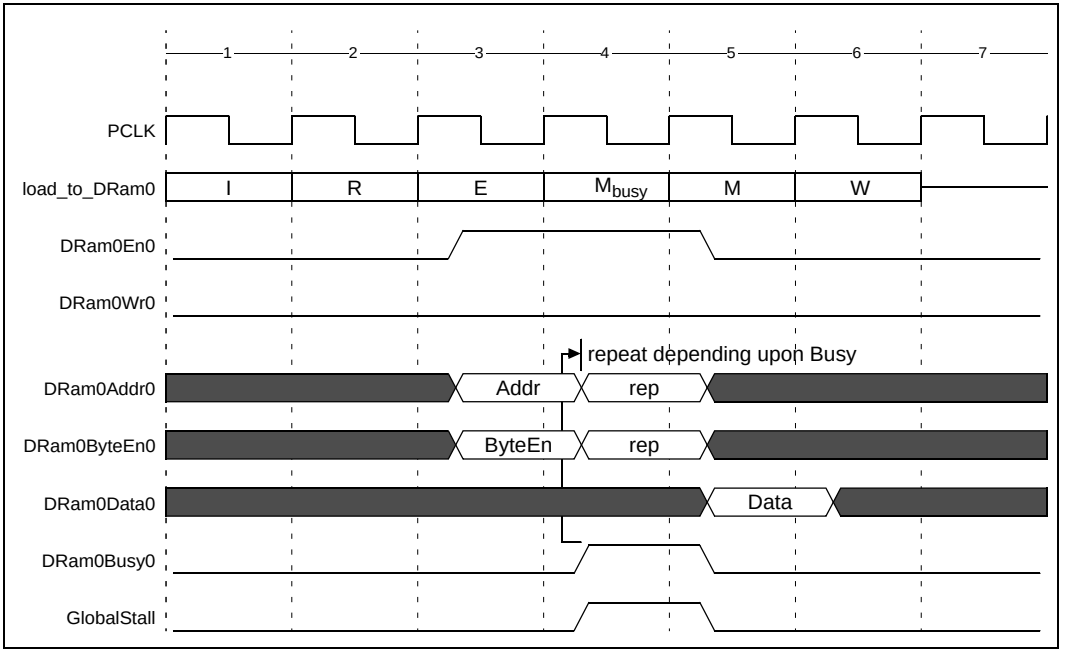


Figure 18–81. Data-RAM Load with Busy (5-Stage Pipeline)

To retry the load, the processor must stall its pipeline. This stall is shown in Figure 18–81 by the M_{busy} in cycle 4 and by the assertion of the internal signal `GlobalStall`. Assertion of `GlobalStall` in a cycle indicates that the processor pipeline is stalled during that cycle. Note that the `GlobalStall` signal is not an external data-RAM signal. It is an internal signal and is only shown here to further illustrate the example.

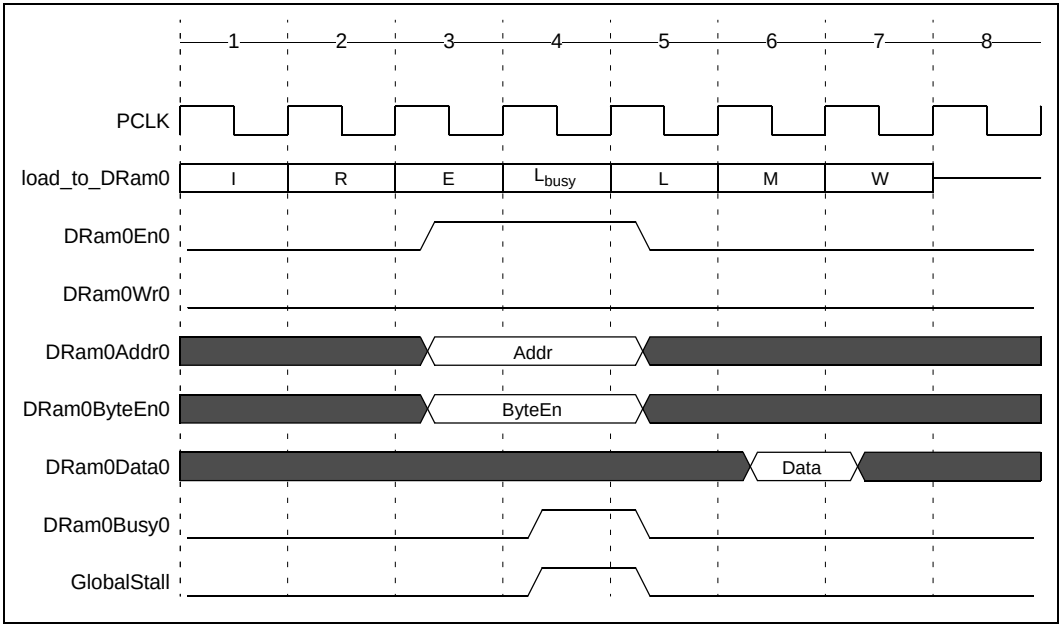


Figure 18–82. Data-RAM Load with Busy (7-Stage Pipeline)

Figure 18–82 shows the data-RAM load-busy assertion case for a processor with a 7-stage pipeline. In this case too, the processor chooses to retry the load upon seeing busy.

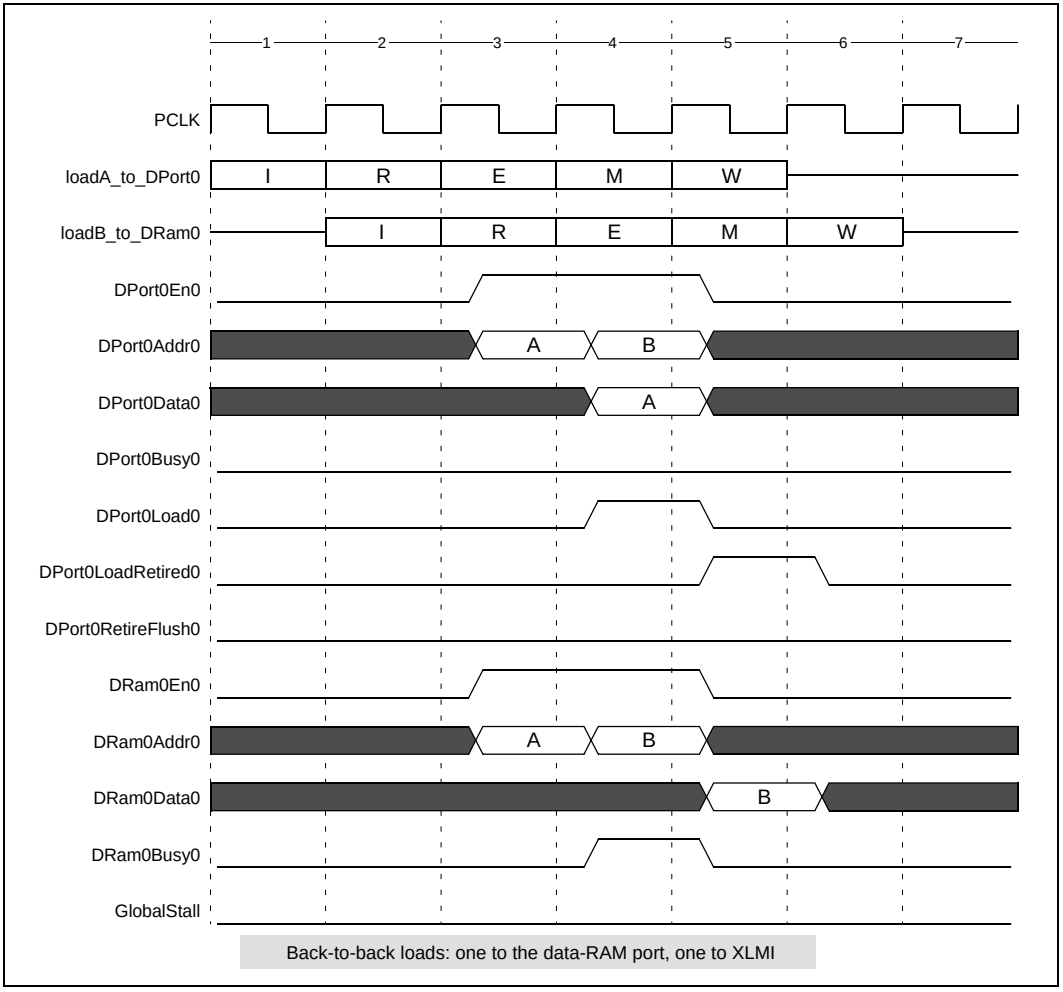


Figure 18–83. Back-to-Back Loads with Busy (5-Stage Pipeline)

Figure 18–83 shows the operation of the data-RAM busy signal when there are back-to-back loads. Several points are demonstrated in the figure:

- No stall occurs during the XLMI load—in cycle 4—even though the data-RAM is asserting its Busy. Restating for the general case, no pipeline stall occurs if Busy of a non-target local memory is asserted.
- The processor asserts the data-RAM address and control signals—in cycle 4—even though the data-RAM is asserting its Busy during this cycle. Again, restating for the general case, the address and control signals from the processor to a local memory will be asserted even in a cycle where the memory's Busy is asserted and the processor will expect either Busy or data back from the memory in the next cycle.

- The data-RAM asserted its Busy in response to the address and enable of cycle 3. The address in the next cycle is completely different because a new transaction has begun. Restating for the general case, in the cycle that a given local memory asserts its Busy, the processor may elect to initiate a different transaction from the one that generated the Busy indication. *Assertion of address and control in one cycle followed by a Busy assertion in the next cycle constitutes a complete transaction.*

Figure 18–84 shows Busy being asserted to a store. In this case, the processor chooses to retry the store in the next cycle, and therefore represents the address data and control.

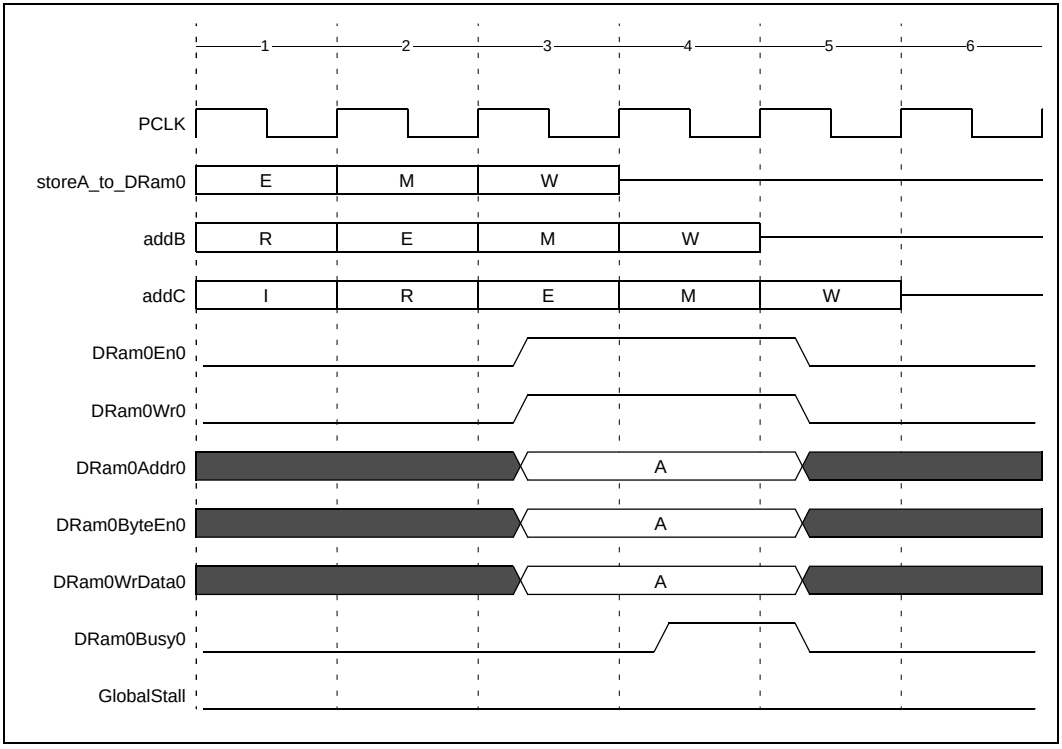


Figure 18–84. Busy to Store (5-Stage Pipeline)

Figure 18–85 shows Busy to a store in the 7-stage pipeline case. As discussed in Appendix B, the write timing is the same whether a 5 or 7-stage pipeline, that is, a read transaction in the cycle after the write (in cycle 5 in Figure 18–84, and in cycle 6 in Figure 18–85) must see the new data.

Note, in both Figure 18–84 and Figure 18–85 the processor pipeline is not stalled (i.e. GlobalStall), even though a store received a Busy response. The processor pipeline only stalls due to stores that receive Busy responses if the processor's store-buffer is

full and there is another pending store in the pipeline. 5-stage configurations have two store-buffers per load/store unit, and 7-stage configurations have three store-buffers per load/store unit.

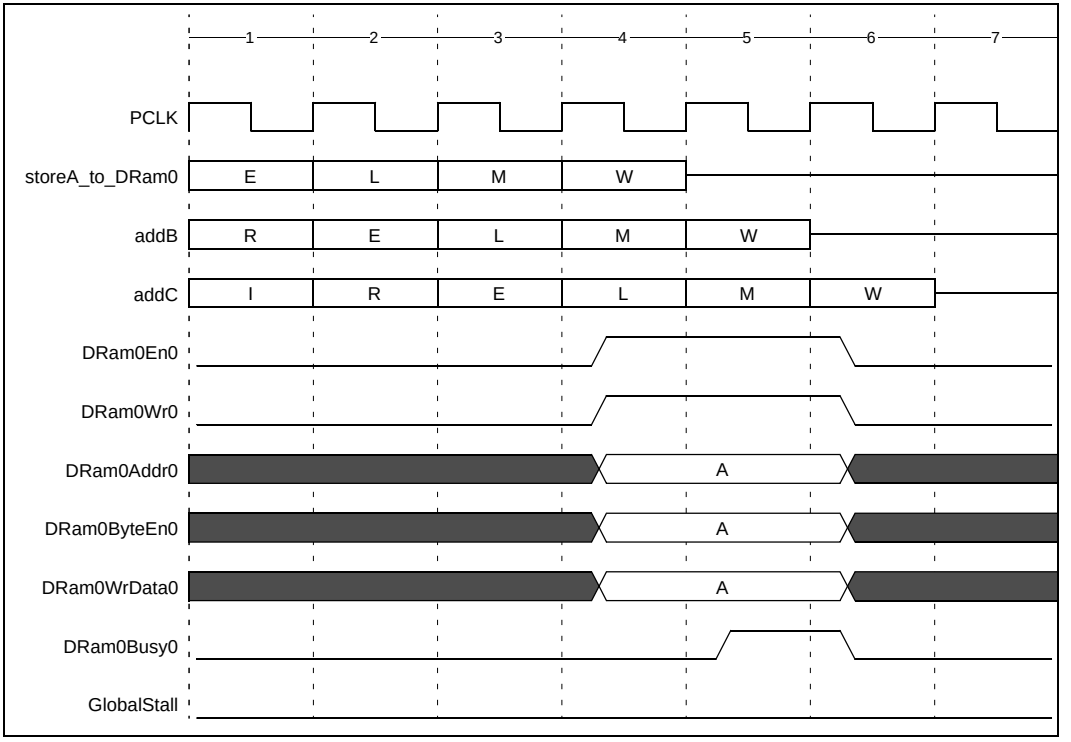


Figure 18–85. Busy to Store (7-Stage Pipeline)

Figure 18–85 shows that Busy acts as a transaction terminate signal. Assertion of DRam0Busy0 during cycle 4 terminates the Store A transaction. Instead of retrying the store, the processor decides to try load D. In general, local memory access is rearbitrated every cycle.

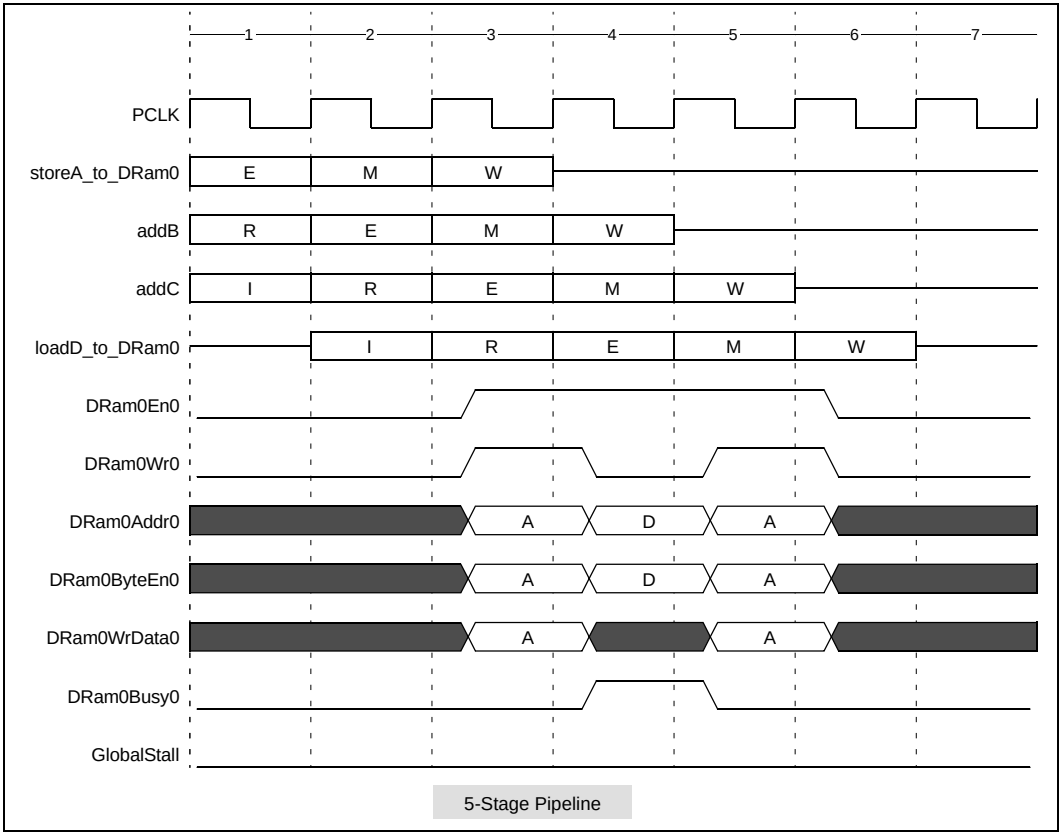


Figure 18–86. Processor Tries Different Transaction After Busy Terminates Store

18.9.3 Inbound PIF to Data RAM Transactions

Inbound-PIF access can be directed at the data-RAM port. Inbound-PIF requests are serviced by a dedicated data/control path that is independent of pipeline, so that inbound-PIF operations can occur without affecting the processor pipeline, thus improving data moving efficiency.

Note: Inbound PIF is independent of Xtensa pipeline. For applications that contain data dependent between inbound PIF requests and Load/Store instructions, software or external agents need to synchronize dependent accesses appropriately.

Figure 18–87 shows that, inbound-PIF operations can happen without affecting the processor pipeline. In this example, inbound-PIF access is directed to DRam0, while a load instruction is directed to DRam1. Since there is no conflict, their addresses are sent at cycle 2 and read datum are back at cycle 4.

Note: If no contentions exist, an inbound-PIF RCW lock request can be processed in parallel with an S32C1I lock request. As a result, two lock operations may each lock their own physical data RAM bank simultaneously.

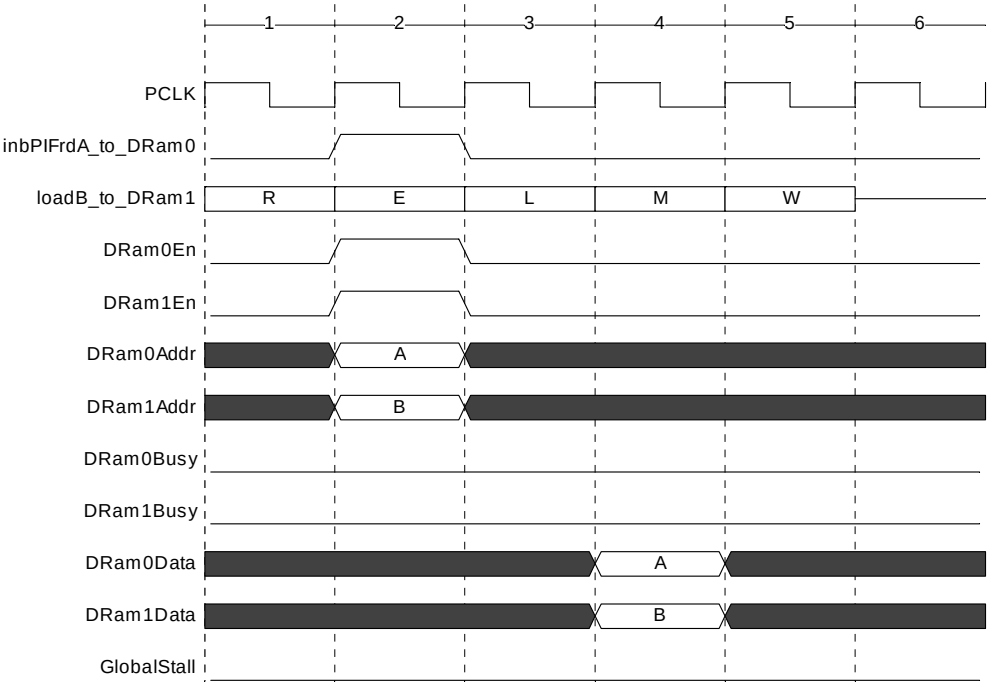


Figure 18–87. Inbound-PIF Read to DRam0, Parallel w/Load from DRam1 (7-Stage Pipeline)

Inbound-PIF transactions directed at the processor's data RAM employ the same Enable/Busy handshake mechanism. From the data RAM's perspective, these inbound-PIF transactions are indistinguishable from loads and stores initiated by the processor. The Xtensa processor arbitrates between inbound-PIF transaction requests and pipeline-initiated accesses to data RAMs with the rules described in Section 18.9.3.

Based on the two-bit PIReqPriority input pins, high-priority (2'b11) inbound-PIF requests get preemptive access to data RAM. Low-priority (2'b00,2'b01,2'b10) inbound-PIF requests get access to data RAM when the memory/bank is not used by load/store units. Low-priority requests eventually graduate to high priority if repeatedly denied access. For example, Table 18–66 shows the allotted data RAM bandwidth for a configuration whose data RAM width matches the PIF data width. Note that the actual percentage of RAM bandwidth achieved is impacted by static and dynamic factors, such as the RAM-

to-PIF width ratio, or how fast the PIF responses can be dispatched, etc. The upper bound of the achievable data RAM bandwidth is the lower of the PIF and data RAM bandwidth.

Note: High-priority inbound PIF requests should be used with caution, as issuing a long sequence of back-to-back high-priority inbound PIF requests may stall the Xtensa processor if they both compete for the same data RAM bank.

Table 18–66. Allocated Bandwidth for Inbound-PIF Request Priorities

PIReqPriority[1:0]	% of Data-RAM Bandwidth Allocated to Inbound PIF
11 (High priority)	100%
10 (Low priority)	Approximately 25%
01 (Low priority)	Approximately 25%
00 (Low priority)	Approximately 25%

Notes:

- Configurations with wide ECC/Parity option may utilize mutex to protect read-modify-write operation. The effect of mutex can impact the inbound PIF performance as described in Section 12.3.4 “Inbound-PIF Read Modify Write Operations”, thus changing the effective bandwidth.
- The data RAM bandwidth is shared by multiple agents. When two agents of the same priority have an access conflict to the same bank, the arbiter will grant their accesses in a rotating manner, resulting in a change of the effective bandwidth each agent receives.

18.9.4 Load/Store Transaction in FLIX Packets

The Xtensa LX7 processor can be configured with either one or two load/store units with multiple local memory banks. As described in Section 18.5, when two load/store units are configured, the C-Box can be configured to resolve conflicts and connect each local memory to the corresponding local-memory (bank) ports.

In general, different local-memory ports operate completely independent. Each port can be configured to have its own busy signal to throttle throughput. When multiple load/store units are configured, load and store instructions can be bundled into FLIX. For bundled loads, the two load operations (within one FLIX instruction) flow through the processor’s pipeline as one, but are illustrated as two pipelines moving in lockstep in the timing diagrams of Figure 18–88. It shows if a memory access causes a global stall, the entire FLIX instruction containing the two load operations also stalls.

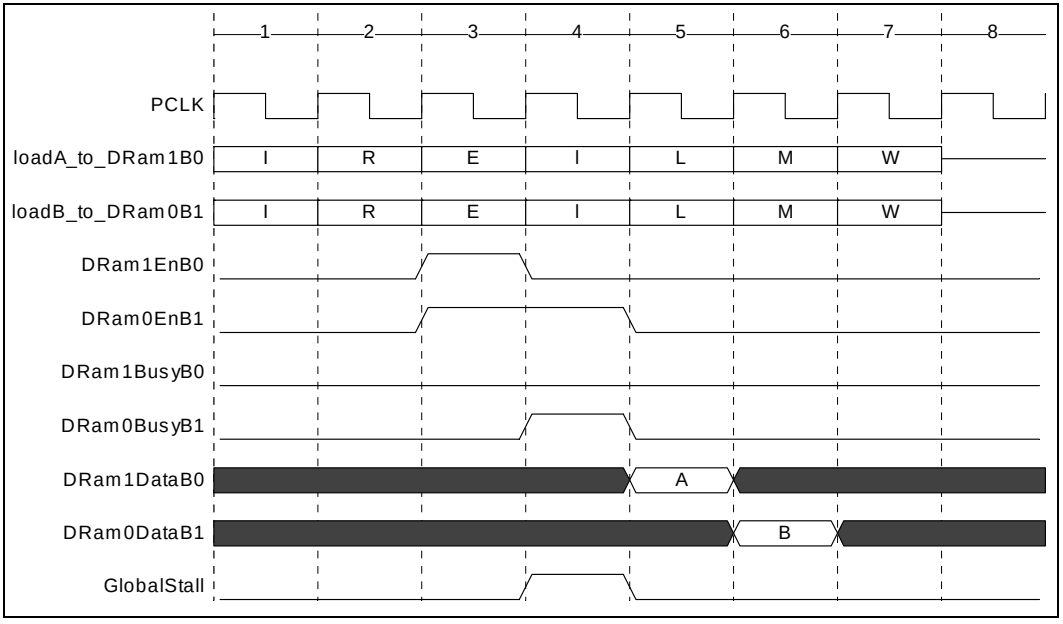


Figure 18–88. Stall of Multiple Loads from Multiple Load/Store Units

Figure 18–88 demonstrates two points with regard to enables:

- After Busy is negated, the associated enable for *that* transaction disappears.
- When enable is first asserted—that is, during the E stage of the load instruction—the enables of all local memories are simultaneously asserted. However, if the target memory asserts its Busy signal, the processor only reasserts the enable for that target memory.

By chance, the loadA transaction in Figure 18–88 completes (with respect to the local memory) before the loadB transaction. It could well have been the other way. Because a Busy signal can be arbitrarily long, load ordering between different load/store units is not guaranteed. Also note that there is no connection between the return of data by the local memory and when the processor pipeline resumes or retires the load.

For stores, the absence of a Busy signal during the cycle following assertion of the write signal signifies that the local memory has successfully accepted the data. If the Busy signal is asserted, the processor continues driving the control and address lines if it chooses to retry the store. Figure 18–89 demonstrates that store ordering is not guaranteed among multiple load/store units.

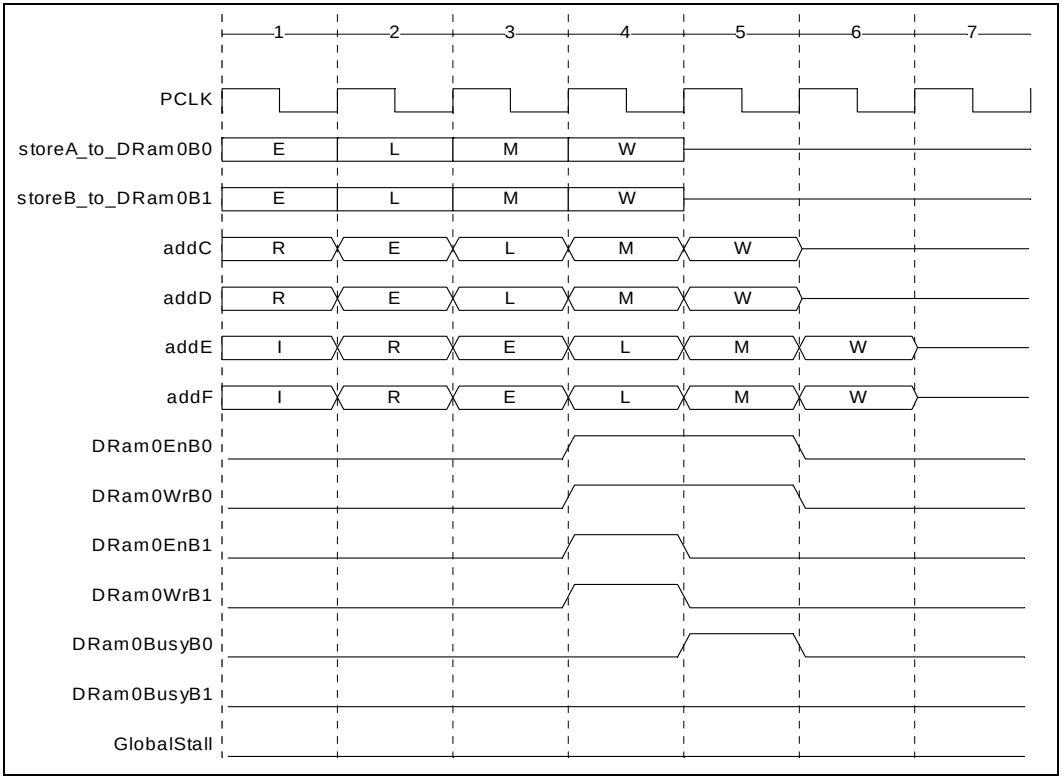


Figure 18–89. Store Ordering with Multiple Load/Store Units (7-Stage Pipeline)

Note that a global stall is only required to avoid a store-buffer overflow. If there is a higher frequency of `Busy` to writes, the store buffer will fill up more quickly.

Note: Memory accesses to overlapping addresses from two different slots in a FLIX instruction must not both be stores, or a `LoadStoreErrorCause` exception will result. In the case of two overlapping addresses from a FLIX bundle consisting of a load and a store, the load will return the previous data and NOT the updated data from the store.

Note: In a multi-slot FLIX instruction bundle, load/store operations directed at instruction ROM and RAM must occupy slot 0 of the instruction bundle, and in the case of two load/store units, it must not be paired with a second, simultaneous, load or store. Otherwise, a load/store exception will occur.

18.9.5 Data RAM Conditional Store Transactions

When the processor issues a Conditional Store transaction, the DRam*n*Lock*m* output pin is present on the data-RAM interface to indicate that an S32C1I transaction is locking the interface. After the lock is granted, this lock signal remains asserted until the S32C1I finishes. Use of the DRam*n*Lock*m* signal is required only when a data-RAM memory is shared among multiple masters in the target system. The DRam*n*Lock*m* signal is initially asserted on the data-RAM interface when an S32C1I instruction reaches the pipeline's E stage. If the data RAM is not the target of the S32C1I instruction, the processor will negate DRam*n*Lock*m* in the following cycle. However, if the data RAM is the target of the S32C1I instruction, the processor will continue to assert DRam*n*Lock*m* until the S32C1I instruction has committed and until all of the processor's store buffers have completely flushed. After the DRam*n* interface is locked, it is essential that the arbiter ensures that no other master accesses the data RAM until DRam*n*Lock*m* is no longer asserted. Note that the ATOMCTL register does not affect the behavior of S32C1I to data Ram in any way.

Figure 18–90 shows an S32C1I transaction to data RAM in which the compare data matches the load data. The comparison of the load data to the SCOMPARE register occurs in cycle 5 and the store to the store buffer occurs in cycle 6. The lock signal remains asserted until both the S32C1I instruction has committed (cycle 5) and until the store buffer has flushed (cycle 7). If the comparison had failed, then the lock signal would be deasserted after the S32C1I instruction had committed.

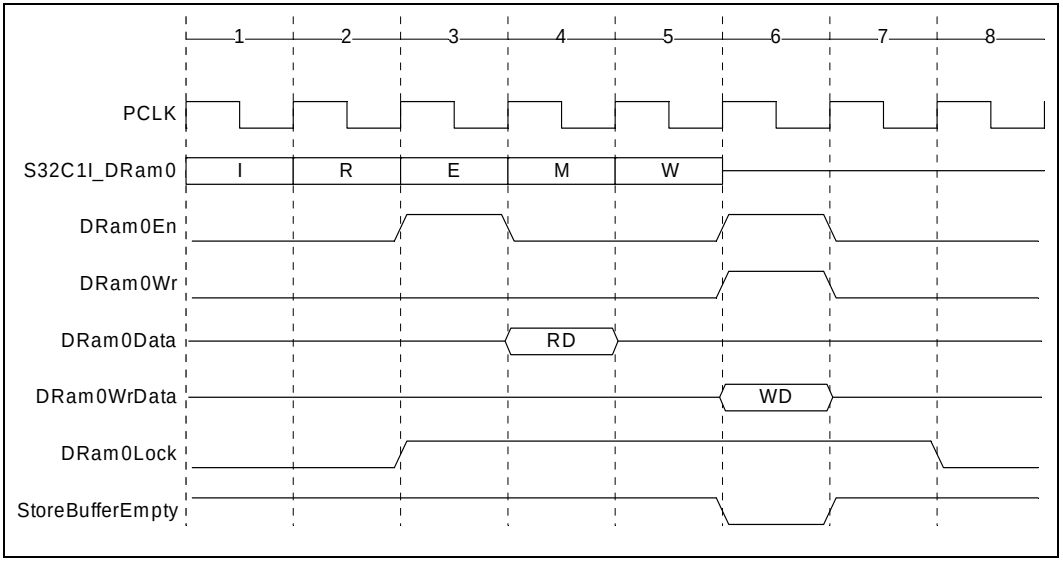


Figure 18–90. S32C1I Transaction to Data RAM

18.9.6 Unaligned Data RAM Transactions

When the Unaligned Handled By Hardware option is configured, the processor will handle unaligned memory accesses by automatically performing multiple transactions to sequential memory locations as needed to the data RAM interface. When an unaligned load or store does not cross a data-RAM-width boundary, only one transaction is required. However, when an unaligned access spans a data-RAM-width boundary, the processor will perform the first portion of the transaction and will replay the instruction to perform the second portion. Any data RAM access spanning a data-RAM-width boundary will always require two transactions separated by a replay.

18.9.7 Data RAM Byte Enables

Data RAM byte enables are valid for both loads and stores. For example, if the data RAM has a 64-bit access width, on a load from Load/Store unit 0 from data RAM 0, the processor will only use the enabled bytes indicated by Dram0ByteEn0[7:0]. Similarly for a store to the data RAM, only the bytes enabled by Dram0ByteEn0[7:0] should be written into the data RAM.

The TIE language allows load or store operations to enable or disable arbitrary bytes of the data transfer. In addition, an inbound PIF interface accepts PIF write transactions with arbitrary byte enables, as defined in the PIF protocol. To support store operations with arbitrary byte enables, a data RAM must be designed to write data only on the designated byte lanes and must not infer operand size or alignment from any combination of asserted byte enables.

18.9.8 Local Memory Arbitration with an External Agent

The C-Box is used to connect the Xtensa processor to data RAMs. From the C-Box's point of view, the connected data RAMs belong to the Xtensa processor exclusively. Sometimes however, a data RAM needs to be shared, for example, between an Xtensa processor and an external DMA controller. To support this, an external agent needs to be designed to multiplex and arbitrate data RAM requests between an Xtensa processor and the other master devices. The external multiplexing allows the external agent to take control of the data RAM and perform read or write operations. Each time the external agent takes control of the data RAM, it must assert DRam*n*Busy*m* on the next cycle to inform the processor that the data RAM was unavailable. This section uses examples to illustrate several design considerations for such use.

Note: An external agent can be designed to support data RAM sharing. It is not used to replace the C-Box.

Figure 18–91 is an example of a load to address A. The Arbiter needs to latch the Address and DRamEn bit when asserting DRamBusy and do a comparison one cycle before de-asserting DRamBusy. In this case comparison succeeds, and data D is made available two cycles later (assuming a 7-stage pipeline).

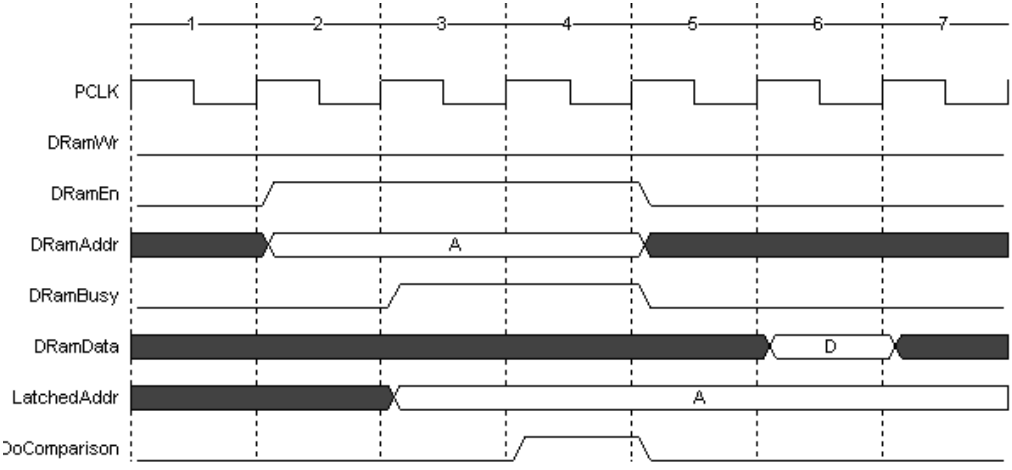


Figure 18–91. Load to Address A

The load to address A in Figure 18–92 shows that a compare one cycle before deasserting DRamBusy fails as the load has changed to a load to address B. The request has changed and as a result returning data D would be incorrect.

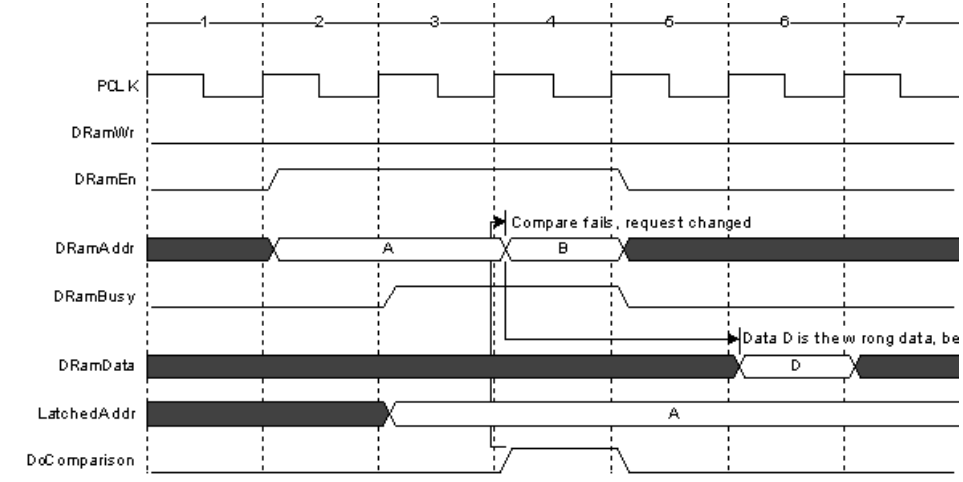


Figure 18–92. Load to Address A, then Address B Incorrectly Handled

The load A followed by Load B shown in Figure 18–93 shows the correct way to handle the situation in Figure 18–92 is to place latch address B after the comparison fails and start a new transaction. Note that busy is not used for extending the memory latency because the return data D for address A will be discarded by the core since the core changed the request address to B.

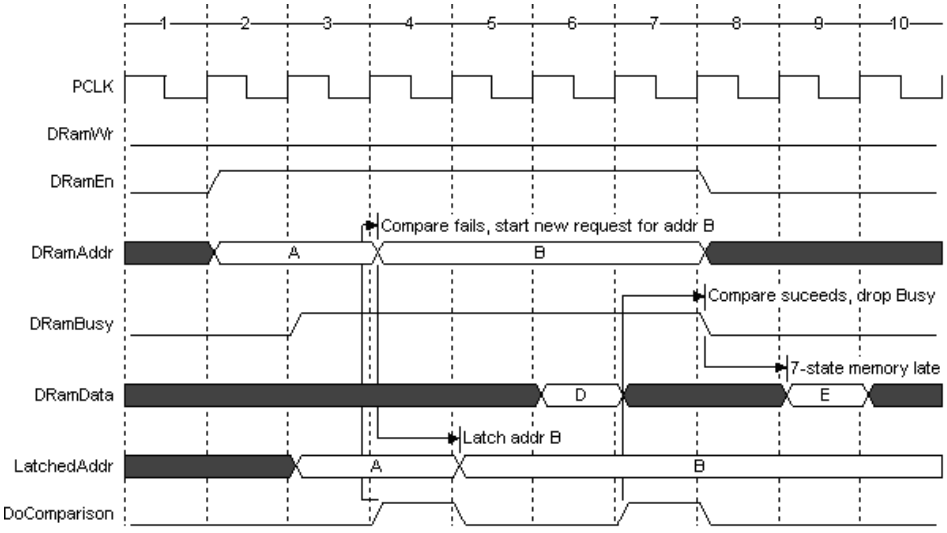


Figure 18–93. Load to Address A, then Address B Correctly Handled

In Figure 18–94, the load transaction becomes a store transaction after asserting busy, which is handled similar to the load transaction in Figure 18–93.

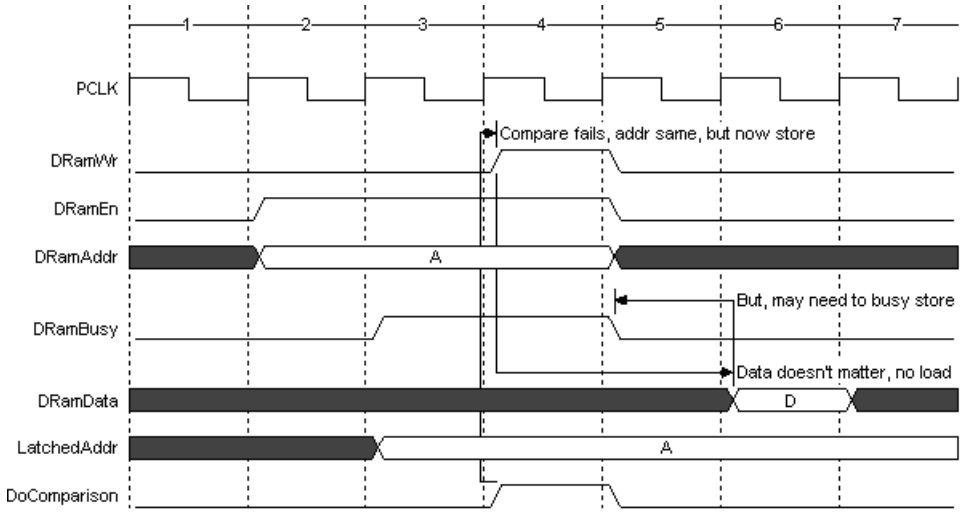


Figure 18–94. Load to Address A, Followed by Store to Address A

Figure 18–95 shows a store transaction that gets a busy signal.

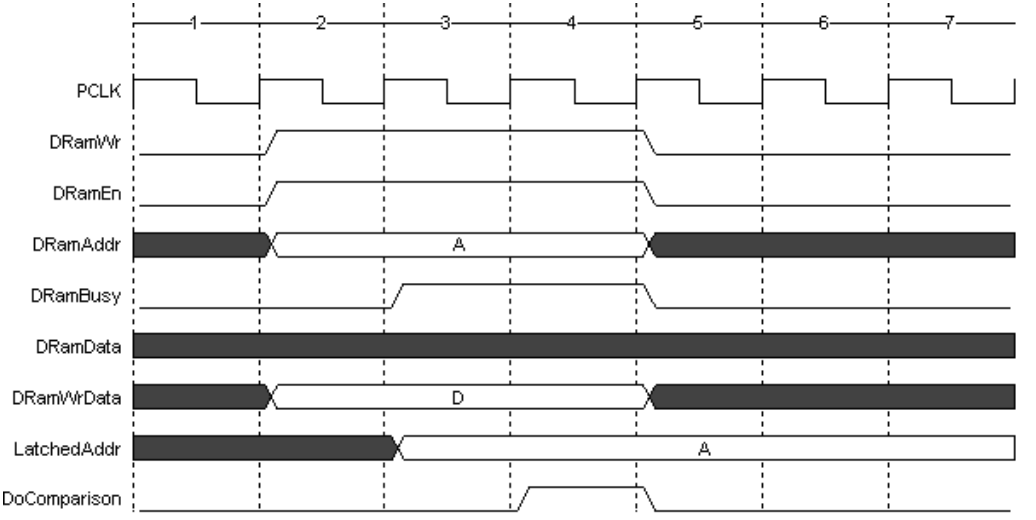


Figure 18–95. Store to Address A with Busy

Figure 18–96 shows a store transaction followed by a load.

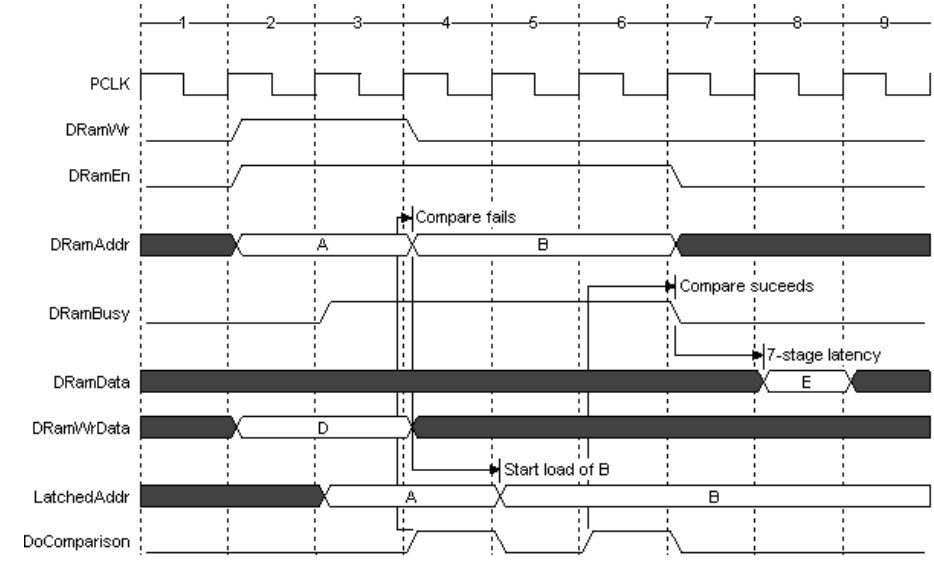


Figure 18–96. Store to Address A, Followed by a Load to Address B

Figure 18–97 shows a load transaction that gets killed after a Busy and never returns.

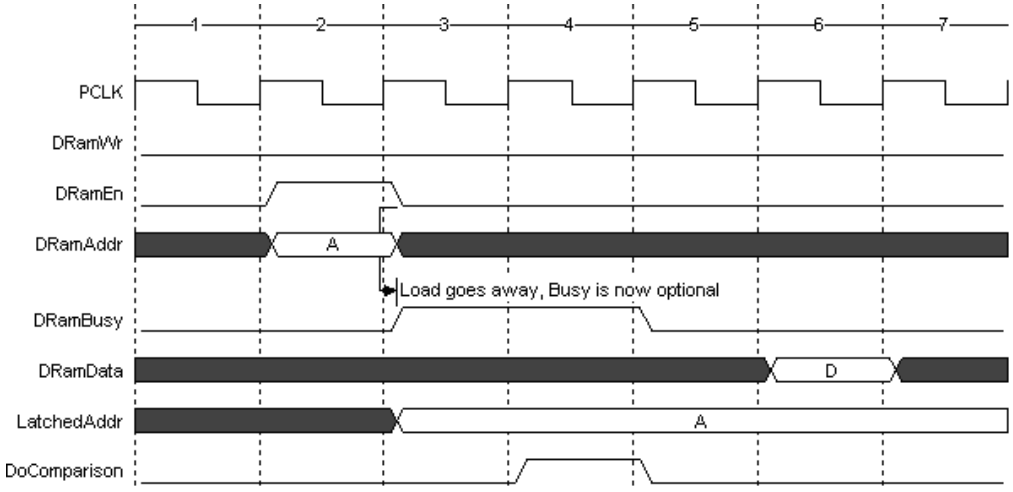


Figure 18–97. Load to Address A gets Killed after Busy

18.10 Exposed Processor Interface for a Customer-Designed C-Box

For two-load/store configurations, the Xtensa built-in C-Box provides a convenient way to directly connect the core to single-ported data RAMs. You may elect to not use Xtensa's built-in C-Box by setting the C-Box to *false*. De-selecting C-Box in two load/store configurations exports the Xtensa core's internal load/store interface to local data memories (data RAM/data ROM). If inbound PIF and two load/store units are configured, the inbound PIF interface to data RAM will also be exported. These exported interfaces allow customers to design an external connection box for their need. The exported interface to data RAM consists of a dedicated signal group for read operations and a dedicated signal group for write operations. They are called split read/write ports. De-selecting C-Box will automatically generate data RAM split read/write ports for each load/store unit (and inbound PIF if inbound PIF can access data RAM) so that the customer-designed connection box can exploit the parallelism between the read and write operations.

Note:

- The C-Box option is not relevant for one load/store unit configuration; the Xtensa core always does the necessary arbitration internally for one load/store unit configurations.
- The Banks option is not relevant and has to be 1 for multiple load/store unit configurations that do not use the C-Box option.
- If C-Box is *false*, the Busy option should be configured to be used by customer-designed external connection box.
- Setting C-Box to *false* in two load/store configurations provides the flexibility for a customer-designed C-Box to exploit the parallelism between read and write operations. It no longer supports the direct connection to a dual-ported memory.

Figure 18–98 shows an example that uses a customer-designed connection box. Up to six independent interfaces are exported. Note that when the data RAM interface is opened, the read and write interface per data RAM of a load/store unit are split and are exported as independent ports. Splitting the read and write ports allows independent loads and stores to be dispatched concurrently. These signals are suffixed with "0" or "1" indicating the identity of the associated load/store unit that initiates the request. Also note that if inbound-PIF can access data RAM, the data RAM read and write interfaces from inbound-PIF are exported and suffixed with "DMA". However, different from load/store units, inbound PIF can only dispatch a single read or write at any given time.

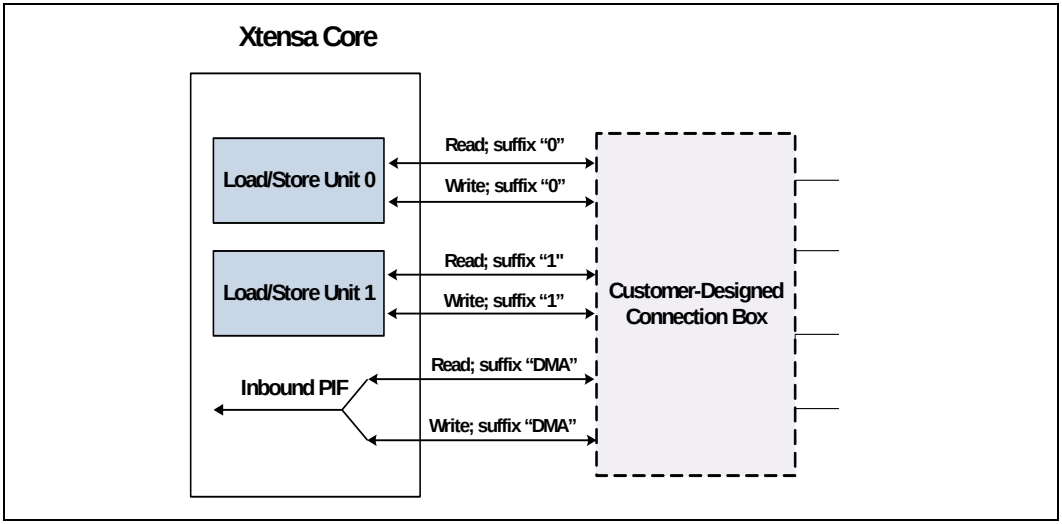


Figure 18–98. Open Data RAM Interface for External Arbiters

Splitting the read and write ports creates a data RAM interface that is different from Table 10–20. These signals, as shown in the following table, are to be used by a customer-designed connection box.

Table 18–67. Data RAM Interface for Split Read/Write Ports

Name (configured width)	I/O	Description	Notes
Read Interface			
DRam*n*En*m*[1,2]	Output	Data RAM read enable. Indicates that a LS or the inbound_PIF is issuing a read	This is asserted only during a read request.
DRam*n*Addr*m* [drambyteaddrwidth-1:dramaccessoffset][1,2,3,6]	Output	Data RAM address lines for a given LS or the inbound_PIF.	Needs to be qualified by DRam*n*En*m*.
DRam*n*ByteEn*m* [drambytes-1:0][1,2,5]	Output	Data RAM read byte enables for a given LS or the inbound_PIF.	Needs to be qualified by DRam*n*En*m*.
DRam*n*Data*m* [dramwidth-1:0][1,2,4]	Input	Data read from data RAM for a given LS or the inbound_PIF.	
DRam*n*CheckData*m* [dramcwidth-1:0][1,7]	Input	Error check bits from data RAM for a given LS or the inbound_PIF	Requires data RAM parity or data RAM ECC
DRam*n*Busy*m*[1,2]	Input	Data RAM read busy signal. Indicates that data RAM memory was unavailable for reading during the previous cycle.	Requires DRamBusy
Write Interface			

Table 18–67. Data RAM Interface for Split Read/Write Ports (continued)

Name (configured width)	I/O	Description	Notes
DRam*n*Wr*m*[1,2]	Output	Data RAM write signal. Indicates that the given LS or the inbound_PIF is trying to execute a write transaction.	Indicates a write when it is asserted. It doesn't need to be qualified by an enable signal.
DRam*n*WrAddr*m* [drambyteaddrwidth-1:dramaccessoffset][1,2,3,6]	Output	Data RAM write address lines for a given LS or the inbound_PIF.	Needs to be qualified by DRam*n*Wr*m*.
DRam*n*WrByteEn*m* [drambytes-1:0][1,2,5]	Output	Data RAM write byte enables for a given LS or the inbound_PIF.	Needs to be qualified by DRam*n*Wr*m*
DRam*n*WrData*m* [dramwidth-1:0][1,2,4]	Output	Data to be written to data RAM for a given LS or the inbound_PIF.	Needs to be qualified by DRam*n*Wr*m*
DRam*n*CheckWrData*m* [dramcwidth-1:0][1,7]	Output	Error check bits to be written to data RAM for a given LS or the inbound_PIF.	Requires data RAM parity or data RAM ECC
DRam*n*WrBusy*m*[1,2]	Input	Data RAM write busy signal for a given LS or the inbound_PIF. Indicates that data RAM memory was unavailable for writing during the previous cycle.	Requires DRamBusy
Misc Signals			

Table 18–67. Data RAM Interface for Split Read/Write Ports (continued)

Name (configured width)	I/O	Description	Notes
DRam*n*Lock*m*[1,2]	Output	Indicates that an atomic transaction, (e.g. S32C1I or RCW) is locking the data RAM interface.	Requires Conditional Store Synchronization Instruction option.
DmaHighPriority	Output	Indicates that the current Inbound PIF request is high priority or has been promoted to high priority after being declined for several cycles.	Requires InboundPIF configured.

Notes:
1. *n* is the associated data RAM
   - *n*=[0] if the configuration has one data RAM.
   - *n*=[0/1] if the configuration has two data RAMs.
2. *m* can be following suffixes indicating the associated load/store unit or inbound_PIF unit.
   - 0: memory is accessed by the first LSU.
   - 1: memory is accessed by the second LSU.
   - DMA: memory is accessed by the inbound_PIF.
3. drambyteaddrwidth = $\log_2$(data RAM size [in bytes])
4. dramwidth = data RAM access width [in bits]
5. drambytes = dramwidth/8
6. dramaccessoffset = $\log_2$(drambytes)
7. dramcwidth = (dramwidth/8) for parity, (dramwidth*5/8) for ECC

18.10.1 Split Read/Write Port Arbitration

The load/store units and the inbound-PIF could request for the same address in the same cycle. When this happens, the external customer-designed C-Box must arbitrate these simultaneous requests. Its arbitration scheme must comply with the following rules:

1. The external customer-designed C-Box can arbitrarily choose zero, one or more simultaneous requests to decline using the busy mechanism.
2. In the case that the external C-Box is willing to service simultaneous read(s) to the same address, all simultaneous read(s) must get consistent value.
3. In the case that the external C-Box is willing to service simultaneous read(s) and a write to the same address, the write data must be forwarded to all simultaneous read(s) to that same address.
4. Simultaneous writes to the same address must not happen in the same cycle.

Note: To enforce the requirement in item 4, the Xtensa core has internal mechanisms to prevent the load/store units from issuing simultaneous writes to the same address. Between the load/store units and the inbound PIF, higher-order software or hardware techniques for enforcing mutual exclusion, such as Read-Conditional-Write PIF request or the S32C1I instruction, should be used to prevent them from competing for the same address.

To correctly handle the lock operations generated from the S32C1I or the Read-Conditional-Write operation, refer to the following section, Lock on Split Data RAM Ports.

18.10.2 Lock on Split Data RAM Ports

If no C-Box is configured for two-load/store configurations, the Xtensa core exports the data RAM interface with split read/write ports. If the Conditional Store Synchronization Instruction option is selected, a data RAM*n*Lock*m* signal will be generated. It is the external customer-designed C-Box's responsibility to handle the lock requests in an atomic manner. This DRam*n*Lock*m* signal is shared by both read and write interfaces that are suffixed with the same *n* and *m*. This DRam*n*Lock*m* signal is asserted when a load/store unit (*m*=0) is doing a conditional store to DRam*n* or an inbound PIF (for example, *m*=DMA) is doing a read-conditional-write (RCW) operation to DRam*n*.

A conditional store/read-conditional-write transaction consists of 1) phase one - a read request with the lock asserted, and 2) phase two - an internal comparison and an optional write request with the lock asserted if the internal comparison succeeds. The Xtensa core starts a conditional store/read-conditional-write transaction by asserting DRam*n*Lock*m* and ends the transaction by deasserting DRam*n*Lock*m*.

In phase one, the address to lock is specified on the read interface using DRam*n*Addr*m* *and* DRam*n*ByteEn*m* (they need to be qualified by DRam*n*En*m*). The customer-designed C-Box decides (based on its own arbitration scheme) whether to accept this lock read request with the lock asserted. It may use the DRam*n*Busy*m* to decline this read request temporarily for one or multiple cycles. The lack of a busy signal in the cycle next to the lock read request is interpreted by the Xtensa core as an acceptance of the lock for the conditional-store/read-conditional-write transaction. When an address is locked, the customer-designed C-Box must decline any other requests to the same address through out the accepted conditional-store/read-conditional-write transaction, thus keeping the service to the granted transaction atomic.

Phase two of a conditional-store/read-conditional-write transaction comprises of an internal comparison and an optional lock write request with the lock asserted. After the read request is accepted in phase one, the read data should be returned by the customer-designed C-Box through the DRam*n*Data*m* bus. It will then be used by the Xtensa core for an internal comparison. If the comparison mismatches, no subsequent write will be sent and the Xtensa core will deassert the DRam*n*Lock*m* to indicate the end of a transaction. If the comparison succeeds, a subsequent write request to the same address will occur for the write interface - DRam*n*WrAddr*m* and DRam*n*WrData*m* (they need to be qualified by DRam*n*Wr*m*). This write request may also be temporarily busied by the customer-designed C-Box using DRam*n*WrBusy*m*. Note that, if a write request with a lock asserted occurs, the customer-designed C-Box must accept it eventually so as to mark the end of the second phase of the conditional store/read-conditional-write. Then the Xtensa core deasserts the DRamnLockm to complete a conditional-store/read-conditional-write transaction.

Note: DRam*n*Lock*m* is asserted by the Xtensa core throughout the conditional store/read-conditional-write transaction so that it distinguishes the read/write during a lock from an ordinary read/write.

18.10.3 Inbound PIF Requests on Split Data RAM Ports

If the inbound PIF option is configured, a Xtensa core may accept inbound PIF access to data RAMs. When C-Box option is false for two-load/store configurations, the data RAMs will have a split read/write interface to service inbound PIF requests one at a time.

Inbound PIF requests that read from data RAMs will present on the data RAM read interface - DRam*n*EnDMA, DRam*n*AddrDMA, DRam*n*ByteEnDMA. DRam*n*AddrDMA and DRam*n*ByteEnDMA need to be qualified by the DRam*n*EnDMA. The customer-designed C-Box shall return the read data on DRam*n*DataDMA. If it is not able to service the read request immediately, the customer-designed C-Box may drive the DRam*n*BusyDMA signal to 1 in the next cycle to temporarily decline the read service.

Inbound PIF requests that write to data RAMs will present on the data RAM write interface -DRam*n*WrDMA, DRam*n*WrAddrDMA, DRam*n*WrDataDMA. The DRam*n*WrAddrDMA and DRam*n*WrDataDMA need to be qualified by the DRam*n*WrDMA. Note that, for split read/write port the DRam*n*W*r*m is independent from DRam*n*En*m*. If it is not able to service the write request immediately, the customer-designed C-Box may drive the DRam*n*WrBusyDMA signal to 1 in the next cycle to temporarily decline the write service.

An auxiliary DmaHighPriority signal is exported on the Xtensa interface. This signal is asserted under the following occasions: 1. the current inbound PIF request has high priority specified by PIReqPriority[1:0]; 2. or the current inbound PIF has been declined for several cycles so that the request has been promoted to high priority. The customer-designed C-Box may choose to use this auxiliary signal in its arbitration scheme so as to give priority to inbound PIF requests of above types. Because inbound PIF requests are serviced one at a time, this DmaHighPriority is shared between the data RAM read and write interfaces.

18.11 Data RAM Memory Integrity

The data RAM can optionally be configured with memory parity or error-correcting code (ECC). Extra memory bits are read from and written to the RAM and are used to detect errors that may occur on the cache data (and to correct some errors). The parity option allows the processor to detect single-bit errors. ECC allows the processor to detect and correct single-bit errors and to detect double-bit errors.

If an uncorrectable memory error is detected on a data-RAM access, an exception is signalled if any byte of the data-RAM data is used by the load which initiated the access. The exception occurs in the W-stage of that instruction, just like any other exception.

When a correctable memory error is detected on a data-RAM access, the behavior of the processor depends on the Data Exception (DataExc) field of the Memory Error Status Register (MESR). If a correctable memory error is detected and the DataExc bit is set, the processor will take an exception. If a correctable memory error is detected and the DataExc bit is not set, the processor will correct the error in hardware and replay the instruction. Following the replay, the data access will proceed using the corrected data.

Note: When the processor's ECC logic corrects a memory error, the erroneous value in memory is not automatically corrected. Only the value provided to the processor through the ECC logic is correct. To correct the stored value in memory, the processor must write the correct data to the appropriate memory location.

If a memory error is detected on a load from DRAM coming from inbound DMA, there are two possible situations:

- When an uncorrectable error is detected, the PIF will return a bus error to the external master.
- When a correctable error is detected, it will be corrected on the fly, but the erroneous value will still reside in memory.

In both cases, the memory error information is not logged internally in the processor.

18.12 Support for Multiple Instruction and Data RAMs

The Xtensa processor allows the designer to configure as many as two instruction RAM ports and two data RAM ports. Each type of these RAMs can be independently configured to use the Busy mechanism. The same type of RAM must have the same Busy configuration. The port configured with Busy has its own Busy signal that operates independently of the Busy signals on other ports.

Requests to processors with inbound-PIF can interact with local RAM while the other local RAMs are being used by the processor.

Note:

- If two instruction-RAM ports are configured, they must both be configured with the same busy and inbound-PIF options. Either both instruction RAM ports have a busy input or they both do not. Either they both have inbound-PIF functionality or they both do not. The same is true for the data-RAM ports.
- Local RAM ports can not be configured to have overlapping addresses.

19. Integrated DMA (iDMA)

The iDMA engine, a single-channel integrated DMA engine, can optionally be configured for the Xtensa processor. iDMA allows for fast data movement between the AXI Bridge and the local data memories and supports data movement between external memory and data RAM and between data RAM and data RAM.

All iDMA operations are controlled by the Xtensa processor through DMA registers and DMA descriptors, which are data structures residing in data RAM. The typical way to operate the iDMA is to prepare descriptors in data RAM and instruct the iDMA to fetch and run a set of descriptors under the control of the iDMA registers. You can check the iDMA status by reading the iDMA status registers.

Each descriptor is a DMA command specifying the essential data movement parameters, for example, the data movement direction and the amount of bytes to transfer, etc. Details about the DMA descriptors are in Section 19.2 “DMA Descriptor Formats”.

The iDMA registers provide control and status reporting. Use the iDMA control registers to specify the parameters of the data movement, for example, how many descriptors to run, what the allowed AXI access block sizes are, how many outstanding bus requests are allowed, etc. You can read the iDMA status registers to poll the iDMA status (IDLE, BUSY, DONE, ERROR, etc.), and diagnose the iDMA errors. Details about the iDMA registers are in Section 19.3 “iDMA Registers”.

iDMA can also be configured with interrupts that notify the Xtensa processor if a particular descriptor finishes or when an error occurs.

Section 19.1 summarizes the key features of iDMA; details of the DMA descriptor formats are in Section 19.2, and the usage of the iDMA registers are in Section 19.3.

Notes:

- iDMA does not support external-to-external memory transfers, data movement to/from the I/O port, (for example, FIFO), control registers, core register, cache, or other non-memory devices.
- The data transfer direction should be consistent throughout each DMA command. Data transfers should not cross the data RAM boundary.
- Because iDMA has limited buffering capability, the source region and destination region should not overlap. Data transferring between the overlapped regions may cause data corruption. Software should provide prevention by checking the descriptor.

19.1 iDMA Features

This section summarizes the key features of the iDMA design. For details about the descriptor formats, refer to Section 19.2 “DMA Descriptor Formats”. For the usage of iDMA registers, refer to Section 19.3 “iDMA Registers”.

19.1.1 1D and 2D Transaction

iDMA supports both 1D and 2D transactions. A 1D transfer starts from a given address and moves a row of continuous datum from source to destination up to a specified number of bytes. A 2D transfer consists of a series of 1D transfers with source/destination pitches that are used to define the distance from two adjacent source/destination rows.

19.1.2 DMA Descriptors

iDMA executes the DMA commands in the format of DMA descriptors. Descriptors are stored in the data RAM. A 1D transfer is described by a 128-bit descriptor. A 2D transfer is described by a 256-bit descriptor. Descriptor formats for the 1D and 2D transfer is described in Section 19.2.1 “1D Descriptor” and Section 19.2.2 “2D Descriptor”, respectively.

19.1.3 Organization of DMA Descriptors

Typically, iDMA fetches and processes the descriptors in a linear manner. You can use an additional special JUMP command (Section 19.2.3 “JUMP Command”) to alter iDMA’s linear fetch behavior.

As Figure 19–99 shows, you can place the JUMP command along with the 1D/2D descriptors to change the iDMA execution flow. Diagram A (on the left) shows that a JUMP command is placed at the tail of a descriptor list. That JUMP points back to the head of the descriptor list. Such a configuration simulates a circular buffer. Diagram B, (on the right) shows that a JUMP command is placed at the tail of the descriptor list on the left. It points to a descriptor list on the right. Such a setup simulates a task list.

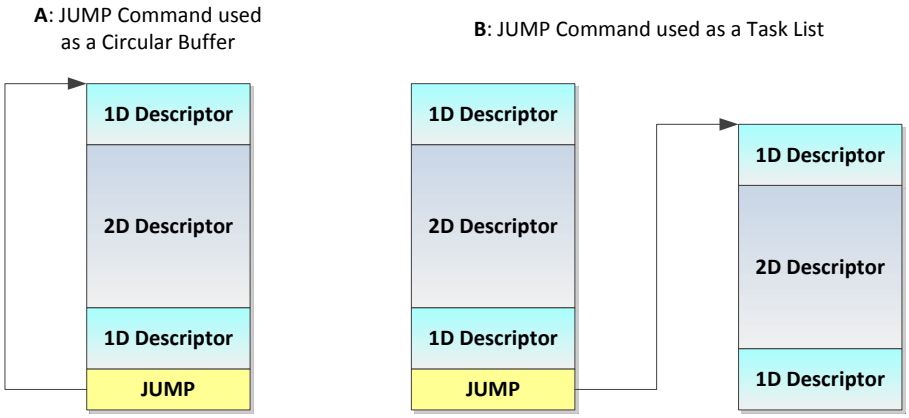


Figure 19–99. Usage of iDMA Descriptors

19.1.4 Flexible Start of iDMA

To start iDMA, programmers have the flexibility to instruct the iDMA to either start fetching descriptors from a new location or to continue from the current descriptor stream (refer to Section 19.3.7 “Control Register” and Section 19.3.4 “Start Address Register”). Programers can also specify the amount of 1D/2D descriptors to run as described in Section 19.3.6 “Number of Descriptors Increment Register”. iDMA keeps track of how many 1D/2D descriptors remain (see Section 19.3.5 “Number of Descriptor Register”) and stops when there are no longer any descriptors. Programmers can add new descriptors into a descriptor queue while the DMA is in progress.

19.1.5 Unaligned Data Transfer

The iDMA does not require transfers to be aligned to the AXI block size or local memory width. The transfer granularity is byte. The start/destination address can be aligned to any byte. The iDMA can automatically shift and align the data accordingly.

19.1.6 Configurable Buffering

The iDMA engine is able to throttle the throughput for busy conditions. It has configurable internal AXI request and response buffers to trade off between area and network congestion. For details of configurability of buffer depth, refer to Section 19.1.15 “Configurability”.

19.1.7 AXI Transactions

iDMA has its own master port. Using a dedicated AXI port allows the system to throttle the DMA throughput independently of the Xtensa processor's external AXI port.[1] (see Section 28.1 "Separate AXI Master Port for iDMA").

iDMA can decompose a large piece of data into multiple AXI transactions. It determines the appropriate AXI transaction type automatically. Use the iDMA setting register to specify the maximum allowed block size and maximum number of outstanding AXI requests.

Descriptors can be set to high or low priority. Bus transactions caused by high-priority descriptors will have the QoS set to high during an AXI transaction.

If the iDMA's access to local data RAM has no conflict with the pipeline/AXI slave port, that access can be serviced immediately. In cases when the iDMA's access to local data RAM is continuously denied due to conflict, iDMA will elevate the priority to the local data RAM, thus allowing it to be granted quickly.

19.1.8 Ordering

A configuration option determines the ordering characteristics of iDMA requests. The default option is to issue all requests with the same request ID. The AXI protocol requires responses to be returned in-order for the requests issued with the same ID.

iDMA's operations are independent of the processor pipeline, in that there is no order relationship between the two. Any synchronization between the pipeline, AXI slave port and iDMA must be handled by software.

19.1.9 Status Reporting

The program can poll the status of iDMA by reading the status registers (Section 19.3.9 "Status Register"). Controlled by the descriptor (Section 19.2.4 "Control Word"), programmers can also choose to interrupt the Xtensa processor when a descriptor completes (i.e.,after you have received all memory transaction acks).

iDMA can also report errors using interrupts. The software can then read the error codes in the status register to get further information.

1. The Xtensa LX7 processor requires the AXI bus option to be selected with iDMA.

19.1.10 Synchronization

The iDMA supports a trigger interface that consists of TrigIn_iDMA(input) and TrigOut_iDMA(output) to allow internal DMA operations to synchronize with external logic. An iDMA descriptor can be programmed to assert TrigOut_iDMA on completion. A descriptor can also be programmed to wait on TrigIn_iDMA before initiation. These two facilities can be used in a variety of ways to synchronize data movement between iDMA units on multiple processors. An example use is a producer-consumer model where processor A sets up its last descriptor to send TrigOut_iDMA, which is connected to processor B's TrigIn_iDMA. In this example, processor B sets up a descriptor to wait on TrigIn_iDMA to start.

19.1.11 Prefetch DMA Descriptors

To improve the performance, a DMA transaction is partitioned into multiple stages. For example, while iDMA is transferring data for the current descriptor, the next descriptor can be fetched from the data RAM. iDMA starts to execute the next descriptor after the current descriptor finishes (i.e., has received all memory transaction acks).

19.1.12 Memory Protection

Before iDMA initiates DMA transactions, it performs a memory protection lookup from the Memory Protection Unit (MPU). This prevents the protected memory region from being accessed by iDMA that does not have privileges. Each 1D/2D descriptor is expected to describe a range that entirely fits in an MPU entry. At the beginning of a DMA row transfer, iDMA may need to do a MPU look up to check that the start and end fit in the MPU memory protection region. iDMA will then transfer data without any MPU query throughout the whole row. Programmers should ensure that the MPU setting is consistent throughout iDMA operations. When iDMA detects that a descriptor could overrun the allowed memory region, it stops and reports an error. For detailed information about the MPU, see Section 3.3.

19.1.13 Wide ECC/Parity Support

If the iDMA accessible data RAMs are configured with the wide ECC/Parity option, an iDMA write to data RAM will trigger a read-modify-write operation if any word will only be updated partially, such as write only one byte in a word. The read-modify-write operation takes multiple cycles to complete.

The read-modify-write sequence is carried out in an atomic manner with regard to the other agents that attempt to update the data RAM at the same time. An internal mutex mechanism is deployed to enforce that only one agent can do a read-modify-write at a time when there is potential data hazard.

19.1.14 WAITI

After the Xtensa processor has prepared iDMA descriptors in data RAM, and instructed the iDMA to run those descriptors, the processor will not be able to enter Wait mode until all the iDMA transfers have completed. (See Section 22.5 for a description of Wait mode.)

If iDMA transfers are still in progress when the processor executes the WAITI instruction, the processor will enter wait mode and assert the PWaitMode signal only after all iDMA descriptors are completed.

19.1.15 Configurability

iDMA provides options to configure the following parameters:

- Configurable bus buffer depth (1/2/4/8/16): Adding more bus buffers provides better capability to off-load the bus traffic; this reduces the chance of iDMA running out of buffers, which can cause a temporary bus deny. This option allows trade off between area and performance.
- Configurable number of outstanding 2D rows (2/4/8/16): Allowing more outstanding 2D row requests improves the iDMA bus performance at the cost of area. This option allows trade off between area and performance.

To use iDMA, a system must support the following:

- The iDMA must be configured with AXI master
- The processor must be configured with data RAM
- The processor must configure interrupt types for iDMADone and iDMAErr
- iDMA only supports Little Endian operations
- C-Box must be configured for two load/store configurations

19.2 DMA Descriptor Formats

Setup DMA transfers using the iDMAlib library. Refer to the *Xtensa System Software Reference Manual* for details.

A DMA descriptor is a data structure residing in data RAM. It describes the behavior of a DMA transaction. There are two types of descriptors: 1D and 2D. They are organized as follows.

1. 1D and 2D share the same fields for the first four words that describe how the row transfer should be done.
2. A 2D descriptor has four extra words that contain information only relevant to 2D transfer.

A special 1-word JUMP command can be placed between descriptors to alter the execution flow.

iDMA can only access descriptors residing in data RAMs. An entire descriptor should be contained with in a single data RAM, that is, a descriptor should not cross a data RAM boundary. A descriptor should be aligned to a 32bit boundary.

19.2.1 1D Descriptor

The 1D descriptor lists the format for 1D DMA descriptors. It consists of four fields, as shown in Table 19–68; each field is a 32-bit word.

Table 19–68. Descriptor Format for 1D Transfers

Memory Placement in 32b Word	Field Name	Description	Bits per Field
1st	CONTROL	Control information	see Table 19–71
2nd	SRC_START_ADRS	The byte address pointing to the start for the 1D transfer. The byte pointed by this pointer is included in the DMA transfer.	32
3rd	DEST_START_ADRS	The byte address pointing to the start of the destination address. The transferred row will be filled to the destination starting from this byte address.	32
4th	ROW_BYTES	The number of bytes to transfer.	32

19.2.2 2D Descriptor

The 2D descriptor lists the data structure for 2D DMA descriptors, as shown in Table 19–69. It consists of eight fields; each field is a 32-bit word.

Table 19–69. Descriptor Format for 2D Transfers

Memory Placement in 32b Word	Field Name	Description	Bits per Field
1st	CONTROL[1]	Control information	see Table 19–71
2nd	SRC_START_ADRS[1]	The byte address pointing to the start for the 1D transfer. The byte pointed by this pointer is included in the transfer.	32
3rd	DEST_START_ADRS[1]	The byte address pointing to the start of the destination address. The transferred row will be filled to the destination starting from this byte address	32

Table 19–69. Descriptor Format for 2D Transfers (continued)

Memory Placement in 32b Word	Field Name	Description	Bits per Field
4th	ROW_BYTES[1]	The total number bytes in a row	32
5th	SRC_PITCH[2]	The byte address distance from one row to the next for the source	32
6th	DEST_PITCH[2]	The byte address distance from one row to the next for the destination	32
7th	NUM_ROWS[2]	The total number of rows in a 2D transfer	32
8th	Reserved	Reserved. Treated as don't-cares	32

Notes:

1. These words are organized the same as 1D descriptors.
2. The words contain information only relevant to 2D transfers.

19.2.3 JUMP Command

Table 19–70 lists the descriptor format for the 32-bit word JUMP command.

Table 19–70. Descriptor Format for JUMP Command

Memory Placement in Unit of 32b Word	Field Name	Description	Bits per Field
1st	JUMP	JUMP command and the target descriptor	32

19.2.4 Control Word

For 1D and 2D descriptors, the control word of the descriptor consists of the same fields described in Table 19–71.

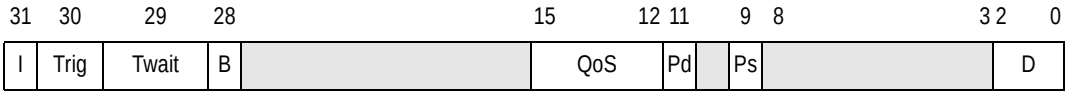


Table 19–71. Control Word

Field	Index	Default	Definition
(D)escriptor	2:0	-	Type of the descriptor: ▪ 3'b011: 1D descriptor ▪ 3'b111: 2D descriptor
Reserved	8:3	-	
Ps	9	-	MPU Privilege for source access: ▪ 1'b0: Supervisor-Privilege ▪ 1'b1: User-privilege ▪ If a DMA transaction reads from bus, Ps is used as a read privilege
Reserved	10		
Pd	11	-	MPU privilege for destination access: ▪ 1'b0: Supervisor-Privilege ▪ 1'b1: User-privilege ▪ If a DMA transaction writes to bus, Pd is used as a write privilege
QoS	15:12	-	Priority: These bits are used when DMA accesses AXI, ▪ 4'b0xxx: Low-priority AXI access ▪ 4'b1xxx: High-priority AXI access ▪ The lower three bits are reserved (don't-care value) Note: Bit 15 of the iDMA Control Word determines the values of the AXI QoS signals. If bit 15 is a zero, the AxQoS signals will be 4'b0000; if bit 15 is a one, the AxQoS signals will be 4'b1100.
Reserved	28:16		
Twait	29	-	Trigger Wait: ▪ When set, the descriptor will wait for HaveTrig (see iDMA Status register) to be set in iDMA's status register

Table 19–71. Control Word

Field	Index	Default	Definition
Trig	30	-	Trigger: When it is set the DMA will send a pulse to the TrigOut_iDMA pin when the descriptor is completely done
I	31	-	Interrupt on successfully completion of descriptor ■ 1'b0: disable ■ 1'b1: enable

Note: The reserved fields are treated as don't-cares.

19.2.5 SRC_START_ADRS

For both 1D and 2D descriptors, SRC_START_ADRS specifies the transfer read start address. It consists of the following field as shown in Table 19–72.

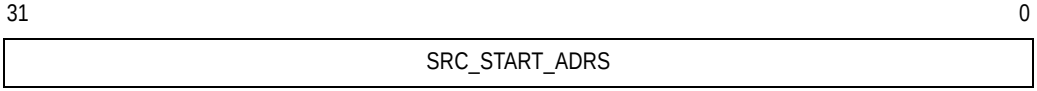


Table 19–72. SRC_START_ADRS

Field	Index	Default	Definition
SRC_START_ADRS	31:0	-	■ The source start address ■ NULL address is an error

19.2.6 DEST_START_ADRS

For both 1D and 2D descriptors, DEST_START_ADRS specifies the transfer write start address. It consists of the following field as shown in Table 19–73.

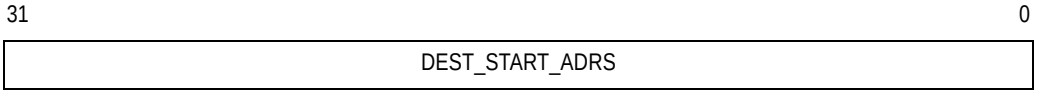


Table 19–73. DEST_START_ADRS

Field	Index	Default	Definition
DEST_START_ADRS	31:0	-	■ The destination start address ■ NULL address is an error

19.2.7 ROW_BYTES

For both 1D and 2D descriptors, ROW_BYTES specifies how many bytes to transfer in a row. It consists of the following field as shown in Table 19–74.

Table 19–74. ROW_BYTES

Field	Index	Default	Definition
ROW_BYTES	31:0	-	▪ The number of bytes to move in a row. It is a signed integer ▪ Negative or zero is treated as an error

19.2.8 SRC_PITCH

For 2D descriptors, SRC_PITCH specifies the distance between two adjacent source rows. It consists of the following field as shown in Table 19–75.

Table 19–75. SRC_PITCH

Field	Index	Default	Definition
SRC_PITCH	31:0	-	▪ The source pitch is a signed integer. When the current row transfer finishes, iDMA uses the current source row's start address plus the SRC_PITCH to calculate the start address of the next row ▪ A negative pitch is treated as an error ▪ Zero is allowed; All the rows will have the same starting source address.

19.2.9 DEST_PITCH

For 2D descriptors, the DEST_PITCH specifies the distance between two adjacent destination rows. It consists of the following field as shown in Table 19–76.

Table 19–76. DEST_PITCH

Field	Index	Default	Definition
DEST_PITCH	31:0	-	■ The destination pitch is a signed integer. ■ When the current row transfer finishes, iDMA uses the current destination row's start address plus the DEST_PITCH to calculate the destination address of the next row ■ A negative pitch is treated as an error ■ Zero is allowed; All the rows will have the same starting destination address.

19.2.10 NUM_ROWS

For 2D descriptors, NUM_ROWS specifies how many rows to transfer. It consists of the following field as shown in Table 19–76.

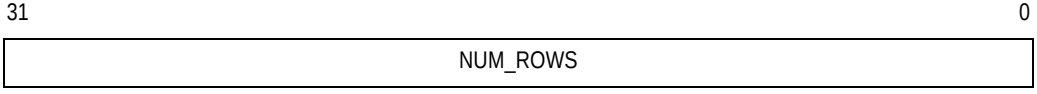


Table 19–77. NUM_ROWS

Field	Index	Default	Definition
NUM_ROWS	31:0	-	■ The number of rows in a 2D transfer is a signed integer. ■ Negative or zero is illegal

19.2.11 JUMP

JUMP command can be used to change the iDMA execution flow. It consists of the following fields as shown in Table 19–78.

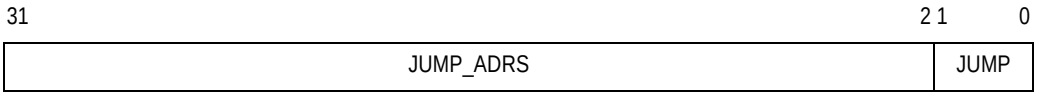


Table 19–78. JUMP

Field	Index	Default	Definition
JUMP	1:0	-	JUMP command: 2'b00: JUMP
JUMP_ADRS	31:2	-	Descriptor address to jump to. The lowest two bits are always 0, because the descriptor and command are word aligned

Typically, iDMA executes descriptors one after another in a linear manner. Using this JUMP command allows iDMA to change the execution flow. Mixing ordinary descriptors with the JUMP command can in effect create a circular buffer or a linked list.

The lowest two bits are used to distinguish a JUMP command from a 1D/2D DMA command:

- 2'b00 means a JUMP
- 2'b11 means a 1D/2D DMA command

19.3 iDMA Registers

Refer to the *Xtensa System Software Reference Manual* for iDMAlib library details.

The processor communicates with iDMA through the WER(write)/RER(read) instructions. Assuming the debugger has privileges, these registers can also be accessible through OCD just like the processor. This section lists the iDMA registers that can be accessed by the processor.

When iDMA is in IDLE mode, the program can initialize iDMA parameters by writing to the Setting/Timeout/DescStartAddr/UserPriv registers. These registers, as described later, provide the runtime parameters for iDMA and should not be changed during DMA transfers.

19.3.1 iDMA Registers Overview

iDMA has several software-accessible registers via RER and WER instructions The iDMA device space includes a supervisor portion and a user portion (defined in Table 19–79).

A special register called ERACCESS is implemented in the Xtensa processor. Bit 1 in ERACCESS specifies if the iDMA user portion is accessible by user code.

The Xtensa processor can access iDMA registers in the following two modes:

- If the Xtensa processor is in privilege mode, the supervisor portion and user portion may be accessed by RER/WER.
- If the Xtensa processor is in user mode, the user portion may be accessed by RER/WER, if and only if, ERACCESS[1] is set.

A PrivilegeViolationCause exception is raised when:

- User code attempts to access the user portion when ERACCESS[1] is clear.
- User code attempts to access the privilege portion.

The following table shows there a couple differences between the supervisor and the user portion: the address assignments are different and the Privilege register is writable only in the supervisor portion and readable only in the user portion.

Table 19–79. iDMA Registers

Register	Supervisor Address	User Address	Description
Settings	0x00110000	0x00910000	iDMA setting register
Timeout	0x00110004	0x00910004	Timeout threshold
DescStartAdrs	0x00110008	0x00910008	Descriptor start address
NumDesc	0x0011000c	0x0091000c	The number of descriptors to process
NumDescIncr	0x00110010	0x00910010	Increment to the number of descriptors
Control	0x00110014	0x00910014	iDMA control register
Privilege	0x00110018	0x00910018	iDMA privilege register. Readable and writable in supervisor portion; Readable only is user portion if ERACCESS=1; Not writable in user portion.
Status	0x00110040	0x00910040	iDMA status register
DescCurrAdrs	0x00110044	0x00910044	The current descriptor address
DescCurrType	0x00110048	0x00910048	The current descriptor type
SrcAdrs	0x0011004c	0x0091004c	Source address from which iDMA is reading from
DestAdrs	0x00110050	0x00910050	Destination address to which iDMA is writing to

19.3.2 Settings Register

The processor can read and write to the Settings register to choose DMA transaction settings.

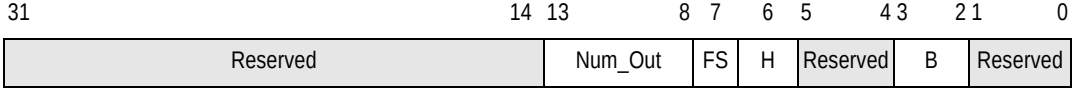


Table 19–80. Settings Register

Field	Index	Default	Definition
Reserved	1:0	-	
B(lock)	3:2	2'b11	Max Block size of DMA: ■ 2'b00: Block 2 ■ 2'b01: Block 4 ■ 2'b10: Block 8 ■ 2'b11: Block 16
Reserved	5:4	-	
H(altable)	6	1'b1	Allow Debug Mode to halt iDMA: ■ 1'b0: disallow Debug Mode to halt iDMA ■ 1'b1: allow Debug Mode to halt iDMA
FS(fetch start)	7	1'b1	When FS is set, iDMA can use DescStartAdrs as the next fetch address
N(um_out)	13:8	6'b000000	Maximum number of outstanding AXI requests allowed simultaneously, within a range of 1-64. The value 6'b000000 is for 64 outstanding. For example, writing 6'b010000 sets the maximum number of outstanding requests to 16.
Reserved	31:14	-	

This setting register allows the program to control how iDMA works, as follows:

- B(lock) controls the maximum allowed AXI request block size.
- N(um_out) controls the allowed maximum number of AXI outstanding requests
- H(altable) controls whether OCDMode can halt iDMA
  - When it is set, iDMA will be put into halt mode during OCDMode. After halting, iDMA maintains its internal states and may be resumed after the OCDMode.

The FS register controls from where iDMA fetches the next descriptor:

- When the IDLE-mode iDMA is enabled, it checks FS; if it is set, iDMA uses DescStartAdrs as the next fetch address. The FS is cleared automatically after it is consumed.
- If the FS bit is cleared, iDMA does not use DescStartAdrs as the next fetch address.

Notes:

- The Xtensa processor should not write FS when iDMA is running, as the behavior is undefined for such cases.
- Before starting DMA transactions, set Num_out according to the buffering capability of the interconnection fabric.

19.3.3 Timeout Register

The processor can read and write to the Timeout register that specifies the timeout limit.

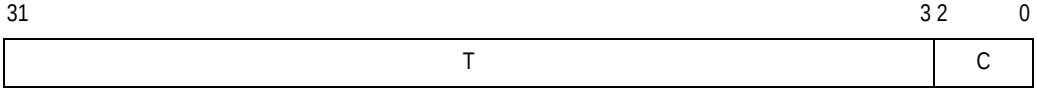


Table 19–81. Timeout Register

Field	Index	Default	Definition
C(locks)	2:0	3'h0	The iDMA's internal timer ticks every C cycles. C specifies the clock cycles that equals to a tick: ■ 3'b000: 1 cycle ■ 3'b001: 2 cycles ■ 3'b010: 4 cycles ■ 3'b011: 8 cycles ■ 3'b100: 16 cycles ■ 3'b101: 32 cycles ■ 3'b110: 64 cycles ■ 3'b111: 128 cycles
T(imeout)	31:3	29'h0	■ The threshold for timeout. When the total ticks for a descriptor reaches T, a timeout error occurs. ■ When T is zero, timeout is disabled.

Use this register to specify timeout ticks and to catch descriptors that run too long.

When timeout is enabled, the timer keeps counting when iDMA is active. The timer is reset when a DMA descriptor finishes. If the timer reaches the timeout threshold, the iDMA stops and interrupts the processor to report a timeout error.

If T(imeout) is zero, the timeout mechanism is disabled.

19.3.4 Start Address Register

The processor can read and write the DescStartAdrs register that specifies the address of the first descriptor.

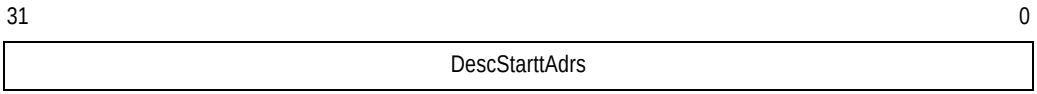


Table 19–82. DescStartAdrs Register

Field	Index	Default	Definition
DescStartAdrs	31:0	32'b0	The start address to the iDMA descriptor. Writing to this register allows the program to specify where to start iDMA. The lowest two bits are hardwired to 0.

Used along with FS(FetchStart) in the Setting register, the DescStartAdrs register allows iDMA to start from an user-specified descriptor address.

19.3.5 Number of Descriptor Register

The processor can read this status register to get iDMA progress. This is a read-only register. Writing to it takes no effect.

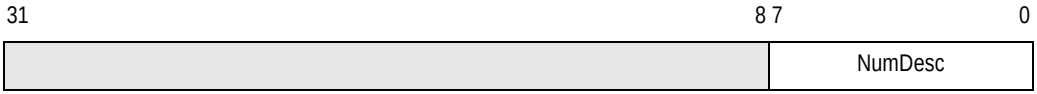


Table 19–83. NumDesc Register

Field	Index	Default	Definition
NumDesc	7:0	8'b0	The number of descriptors remaining to process. This is an unsigned integer.
Reserved	31:8	-	

When iDMA is enabled, it executes the number of descriptors specified in NumDesc. When NumDesc becomes zero, iDMA stops.

NumDesc keeps track of how many (up to 255) descriptors remain to be processed. The processor may read this register to see how many descriptors remain. To increment NumDesc, use the write-only NumDescIncr register (Section 19.3.5 “Number of Descriptor Register”).

NumDesc increases by the amount of *n* when:

- The processor writes *n* to the NumDescIncr register. Thus, if the current value is *m*, the new value with the increase will be *m+n* (see Section 19.3.6).
- If NumDesc overflows the allowed range (255), a NumDesc overflow error is reported and iDMA stops.

NumDesc decreases by 1, when either a 1D or 2D descriptor finishes successfully.

NumDesc will not change when

- A JUMP descriptor is executed

19.3.6 Number of Descriptors Increment Register

The processor can write the NumDescIncr to increment the number of descriptors, which allows the processor to add descriptors. This is a write-only register. The read value is undefined.

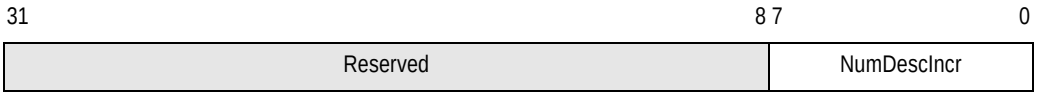


Table 19–84. NumDescIncr Register

Field	Index	Default	Definition
NumDescIncr	7:0		Increments the number of descriptors by this specified amount. This is an unsigned integer. Itis a write-only register.
Reserved	31:8	-	-

The program should keep track of the tail of the descriptor buffer/list and when a new descriptor needs to be added, the program may add it to the tail and update the tail pointer. After new descriptors are added, the program increments NumDesc by writing the delta value into NumDescIncr. NumDesc is then increased by this specified amount.

Writing to NumDescIncr will flush iDMA's descriptor buffers, thus allowing it to re-fetch new descriptors.

19.3.7 Control Register

The processor can read and write the control register to control the start of the iDMA.

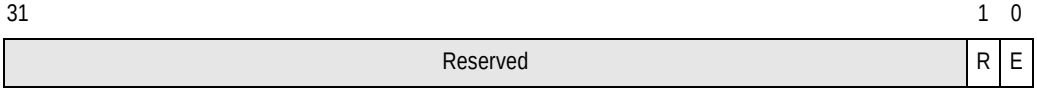


Table 19–85. Control Register

Field	Index	Default	Definition
E(nable)	0	1'b0	Enable iDMA
R(eset)	1	1'b0	When Reset = 1'b1, Reset iDMA
Reserved	31:1	-	Reserved

The processor can write E to enable iDMA, E can be:

- 1'b1: Enables iDMA:
  - If the FS (fetch start) bit is set and iDMA is in IDLE mode (as described in Table 19–87), DescStartAdrs (Section 19.2.6) is used as the address to fetch the next descriptor.
  - Otherwise, iDMA will continue from the current descriptor.
- 1'b0: Halt Mode:
  - iDMA stops fetching new descriptors or making new requests. Some operations in progress may be allowed to finish to allow iDMA to halt cleanly.
  - The processor can check the Run Mode bits in the Status register to make sure iDMA has been halted.
  - iDMA maintains its internal states after being halted. This enables iDMA to resume later, starting from where it left off when it halted. Read data may be held in iDMA's data registers during the halt.

The processor can write 1'b1 to the R(eset) register to soft reset iDMA:

- Write 1'b1 to R to soft reset iDMA. iDMA registers are reset to the default value.
- Write 1'b0 to R to take iDMA out of soft reset.

- The R register is cleared on processor reset, however, initiating a soft reset by writing a 1'b1 to this register does not automatically clear this bit.
- When iDMA runs into an error, use the soft reset mechanism to reset iDMA and the error code. When iDMA is in halt mode, cancel the remaining tasks by applying the soft reset. Do not apply a soft reset unless iDMA is in ERROR or HALT states to avoid potential undefined side effects.

19.3.8 Privilege Register

The processor can read and write the iDMA privilege register of iDMA. To write it, the code must have kernel/supervisor privilege.

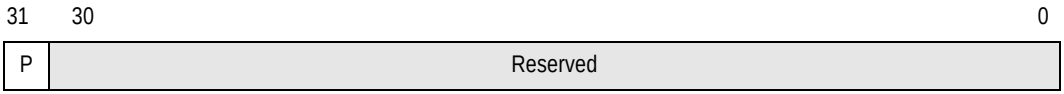


Table 19–86. Privilege Register

Field	Index	Default	Definition
Reserved	30:0	-	
(User)P(riv)	31	1'b0	The privileges of iDMA (for MPU): ■ 1'b0: iDMA is in Supervisor-privilege mode ■ 1'b1: iDMA is in User-privilege mode ■ Reset is supervisor-privilege mode

When iDMA is configured with MPU, iDMA must work within the given privilege. To enforce the security of iDMA operations, there are a few privilege fields designed in iDMA and the descriptor. as follows:

- **Descriptor privilege**: Descriptors have two fields for privilege: one field for source (P_s), and another field for destination (P_d) as described in Table 19–71.
- **iDMA privilege**: This is a dedicated privilege register for iDMA.

The UserPriv bit in the Privilege register is writable ONLY in the supervisor portion. Reading UserPriv is possible in the user portion (when enabled by the special register ERACCESS[1]). When the UserPriv bit is set, iDMA only accesses memory with user privileges. That is, each iDMA fetch/read/write of memory checks that the user has privilege to access that memory location. If user privilege is not available, iDMA reports a privilege error.

The Xtensa processor takes an exception if user code is accessing the supervisor portion, or the user code is accessing the user portion but ERACCESS[1] is clear. iDMA does static and dynamic privilege checks. iDMA does a static privilege check before a descriptor starts as follows:

- When UserPriv=1, iDMA can only execute user-privilege descriptors in which both source privilege and destination privilege are user privilege (Descriptor P_s=1 and P_d=1). Otherwise, iDMA reports an error.
- When UserPriv=0 iDMA can execute both privilege descriptors and non-privilege descriptors.

- When iDMA fetches a descriptor, it queries MPU with the fetch address to learn the boundary of the fetch's protection region. Before a descriptor is executed, iDMA checks if that descriptor's words overrun the fetch protection boundary. It flags an error if the descriptor overruns the protection region.

iDMA does a dynamic check when a descriptor is running, which includes:

- When a row transfer is about to start, iDMA queries the MPU using the source start address and destination start address.
  - If MPU says that access is denied, iDMA reports an error.
  - If MPU grants access, iDMA gets the boundary of the source protection region and the destination protection region. If the source row exceeds the source protection boundary or the destination row exceeds the destination protection boundary, iDMA flags an error.

19.3.9 Status Register

The processor can read the iDMA status register to diagnose the iDMA. This is a read-only register; writing to it has no effect.

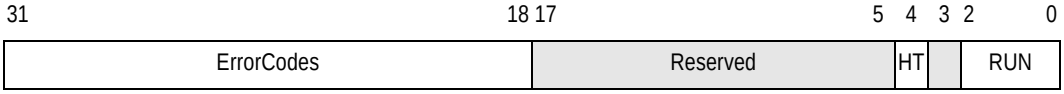


Table 19–87. Status Register

Field	Index	Default	Definition
RunMode	2:0	3'b0	■ 3'b000: IDLE state where iDMA is disabled ■ 3'b001: STANDBY state where iDMA is enabled but cannot start to run descriptor immediately. The reason for STANDBY can be: to wait for clock wakeup from IDLE or to wait for the next descriptor ■ 3'b010: BUSY state where descriptors are being processed or error handling ■ 3'b011: DONE state where there are no more descriptors to process ■ 3'b100: HALT state where iDMA has temporarily stopped on-going transaction and is waiting for an Enable signal to resume. In this mode, data may be held in iDMA's internal buffers. ■ 3'b101: ERROR state. Check ErrorCodes for the error cause. ■ Other value is reserved
Reserved	3	-	
HT(HaveTrigger)	4	1'b0	■ Set by a rising edge of TrigIn_iDMA; ■ Cleared by: the start of a triggered DMA. or by the enabling of iDMA from IDLE state
Reserved	17:5	-	
ErrorCodes	31:18	14'b0	iDMA error code: ■ [31]: Fetch address error ■ [30]: Fetch data error ■ [29]: Read address error ■ [28]: Read data error ■ [27]: Write address error ■ [26]: Write data error ■ [25]: Timeout ■ [24]: Trigger Overflow ■ [23]: NumDesc overflow ■ [22]: Descriptor: unknown command ■ [21]: Descriptor: unsupported transfer direction ■ [20]: Descriptor: bad parameters ■ [19]: Descriptor: null address ■ [18]: Descriptor: privilege violation

iDMA RunMode:

- IDLE: At reset, iDMA is in IDLE state (disabled). iDMA's internal clocks may be turned off in IDLE state to save power. To start DMA operations, write 1'b1 to the Enable bit in the Control register.
- STANDBY/WAKEUP: iDMA goes through a WAKEUP sequence to turn on the clocks when it transits from the IDLE or DONE state to BUSY state.
- BUSY: iDMA is processing a descriptor.
- DONE: The enabled iDMA will park in the DONE state after NumDesc becomes zero. iDMA's internal clocks may be turned off in the DONE state to save power. In the DONE state, iDMA is still active and watching for NumDesc; it wakes up on non-zero NumDesc. Writing 1'b0 to the Enable bit when iDMA is in DONE state will put iDMA in an IDLE state.
- HALT: Writing 1'b0 to the Enable bit when iDMA is in a BUSY state puts iDMA into a HALT state where iDMA temporarily stops all transfers and may be resumed (writing 1'b1 to the Enable bit) later. iDMA will wait until all outstanding bus requests are serviced before entering HALT. In the HALT state, iDMA's internal buffers may still hold data to be transferred.
- ERROR: If an error is detected during operation, iDMA goes to ERROR state and stops. You can read ErrorCodes to get the error cause. In ERROR mode, iDMA expects to be soft reset.

You can either use the completion interrupt on the last descriptor or poll the iDMA status for the DONE state to find out if iDMA has finished its tasks.

HT(HaveTrigger) shows if iDMA receives a TrigIn_iDMA signal. It is set by the rising edge of the TrigIn_iDMA Xtensa interface. It is cleared by the dispatching of a triggered DMA or by the enabling of iDMA from IDLE state. It can hold one trigger event. When the trigger overflows, the trigger overflow error code is set.

Read ErrorCodes to diagnose the type of error, if any. Before re-starting iDMA, clear the error codes using the soft reset mechanism described in Section 19.3.7.

Notes: iDMA will not execute any descriptor(s) until the ErrorCodes are clear in the status register.

iDMA uses the following principles to handle errors:

- For correctable errors, iDMA rectifies them in-flight silently. They will not be logged or reported.
- For uncorrectable or fatal errors, iDMA will stop and report the error. iDMA can also interrupt the processor.

Besides ErrorCodes, the following information is useful when an error happens.

- DescCurrAdrs: When appropriate, DescCurrAdrs records the descriptor address associated with the error. DescCurrAdrs is informational and may not be precise under certain cases. For example, an error is encountered before a descriptor is dispatched to execute.
- SrcAdrs, DestAdrs: You can read these two registers to see the proximity of where the transfer error happens.

Note that the memory transaction error response can happen many cycles after the read/write requests are made, SrcAdrs and DestAdrs does not point to the precise error address.

When iDMA stops for errors, the ErrorCodes are set accordingly as shown in Table 19–88.

Table 19–88. Error Codes

Error Code Bit	Error Type	Causes
31	Fetch address error	The fetch address is not in the data RAM region, fetch access deny, fetch address overrun-mmu-region, etc
30	Fetch data error	Uncorrectable fetch data error
29	Read address error	Read target is changing during a descriptor run., e.g. going from inside of one data RAM to outside, or going from AXI region to non-AXI region. AXI address error, read access deny, read address overrun-mmu-region, etc.
28	Read data error	Uncorrectable read data error, AXI data error, etc.
27	Write address error	Write target is changing during a descriptor run., e.g. going from inside of one data RAM to outside, or going from AXI region to non-AXI region, AXI address error, write access deny, write address overrun-mmu-region, etc.
26	Write data error	Uncorrectable data error during write, AXI data error, etc.
25	Timeout	Descriptor timeout.
24	TriggerOverflow	Asserted when HaveTrigger is set and another TrigIn_iDMA is asserted. In such cases, the trigger overflows.
23	NumDesc overflow	NumDesc overflows
22	Descriptor: unknown command	Unknown descriptor control command

Table 19–88. Error Codes

Error Code Bit	Error Type	Causes
21	Descriptor: unsupported transfer direction	Unsupported transfer direction. The supported transfer directions are: AXI to a data RAM, a data RAM to AXI, or a data RAM to a data RAM.
20	Descriptor: bad parameters	Zero or negative number of rows, zero or negative row bytes, or negative source/destination pitch, etc.
19	Descriptor: null address	A NULL address is used in a descriptor
18	Descriptor: privilege violation	Privilege violation; if user-privilege iDMA attempts to run a supervisor-privilege descriptor.

19.3.10 Current Descriptor Address Register

The processor can read the DescCurrAdrs register to know the current descriptor address. This is a read-only register; writing to it has no effect.

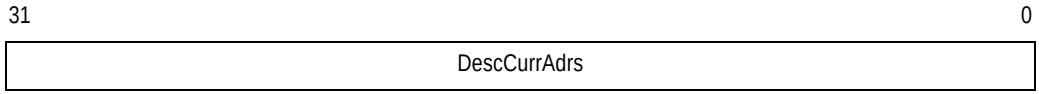


Table 19–89. Current Descriptor Address Register

Field	Index	Default	Definition
DescCurrAdrs	31:0	32'b0	Reflects iDMA progress when read by the processor

Use the DescCurrAdrs register to check the progress of iDMA.Once a new descriptor is dispatched to execute, the DescCurrAdrs is updated to point to the address of that descriptor.

19.3.11 Current Descriptor Type Register

The processor can read the DescCurrType register to know the type of the current descriptor c. This is a read-only register. Writing to it takes no effect.

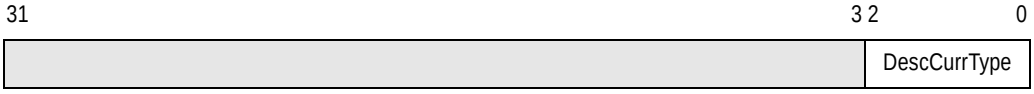


Table 19–90. DescCurrType Register

Field	Index	Default	Definition
DescCurrType	2:0	3'b0	DescCurrType: ■ 3'b011: 1D descriptor ■ 3'b111: 2D descriptor ■ 3'bx00: JUMP.
Reserved	31:3	-	-

Use the DescCurrType register to check the type of the DMA transaction.

Once a new descriptor is dispatched to execute, the DescCurrType is updated to reflect the type of that descriptor

19.3.12 Source Address Register

The processor can read the SrcAdrs register to know the read progress of the current DMA transfer. This is a read-only register. Writing to it takes no effect.

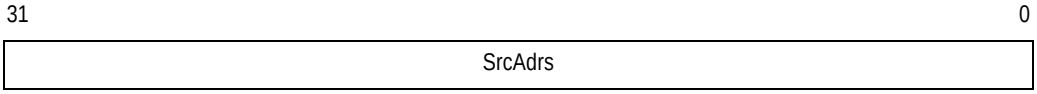


Table 19–91. Source Address Register

Field	Index	Default	Definition
SrcAdrs	31:0	32'b0	The address from which iDMA is reading

Reading the source pointer allows the processor to find the progress of the read transfer. SrcAdrs points to the proximity of where iDMA is reading from. SrcAdrs is valid only when DMA is actively transferring datum.

The SrcAdrs register is aligned to the AXI block/data RAM width in that the offset bits are zero.

Due to the pipeline implementation of iDMA, SrcAdrs is only a read progress indicator for debug purpose.

19.3.13 Destination Address Register

The processor can read the DestAdrsregister to know the write progress of the current DMA transfer. This is a read-only register. Writing to it takes no effect.

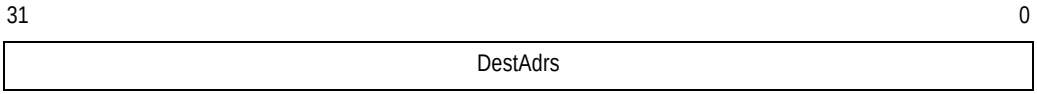


Table 19–92. Destination Address Register

Field	Index	Default	Definition
DestAdrs	31:0	32'b0	The address to which iDMA is writing

Reading DestAdrs allows the processor to find the progress of the write transfer. DestAdrs points to the proximity of where iDMA is writing to and is valid only when iDMA is actively transferring datum.

DestAdrs is aligned to the AXI block/data RAM width in that the offset bits are zero.

Due to the pipeline implementation of iDMA, SrcAdrs is only a write progress indicator for debug purpose.

20. Xtensa Local Memory Interface (XLMI) Port

Note: Cadence recommends using caution before deciding to use the XLMI port on an Xtensa processor as correct implementation of devices attached to XLMI can be a challenging design choice. Alternate interfaces, such as TIE ports, TIE queues, TIE memories, and TIE lookups have simpler and more elegant interfaces. Consult your Applications or Support Engineer for assistance in deciding which interfaces are most appropriate for your SoC design.

The Xtensa Local Memory Interface (XLMI) is a data port that allows for the direct attachment of:

- Hardware peripherals (through memory-mapped I/O registers)
- RAMs
- Inter-processor communication devices (such as FIFOs)

The Xtensa Local Memory Interface (XLMI) has the same low-latency and high-bandwidth properties of the Xtensa local-RAM interface ports described previously, but the XLMI port has special handshake signals that allow attached devices to explicitly deal with speculative loads. Like the Xtensa processor's local data-memory ports, the XLMI port experiences speculative loads from the processor. Speculative loads take data from a memory or peripheral but that data is not consumed within the processor until the processor pipeline retires the corresponding load instruction. If some event (such as a branch or interrupt) kills the load instruction before it is retired, the processor discards and does not consume any data retrieved by that load operation. That same data may be subsequently retrieved when the processor reruns the load operation. Such is the nature of pipelined RISC processors.

Consequently, memories and peripheral devices connected to the XLMI port must be designed to accommodate speculative loads.

Because of these factors, the XLMI port has certain limitations:

- Code cannot be executed from memory attached to the XLMI port. Instruction fetches don't appear on the XLMI port (nor on any of the processor's data-memory ports).
- The `S32C1I` (store 32-bit compare conditional) instruction or Exclusive Access instructions cannot be directed at memory locations mapped to the XLMI port (or to instruction addresses). Such operations will cause the processor to take an exception.
- Like any other block attached to the XLMI port, XLMI-attached memory must not be sensitive to read side effects.
- The MMU cannot perform any address translation for XLMI addresses other than the identity map.

- The Memory Parity/ECC option does not apply to the XLMI port.
- XLMI cannot be configured with multiple load/store units.
- XLMI cannot have multiple banks.
- XLMI cannot have split read/write ports.
- Inbound PIF does not apply to the XLMI port.
- The C-Box option does not apply to the XLMI port.
- Data Memory port widths over 128 bits are not compatible with the XLMI port.
- Unaligned load and store operations directed at the XLMI port trigger an exception, even if the configuration option to add unaligned load/store hardware support is selected.

On-chip peripheral buses such as the AMBA bus are not designed to handle speculative operations and therefore do not interface w*ell with the processor's XLMI port.*

Further, the processor's load/store unit can, in certain situations, promote loads ahead of stores so that the order of loads and stores is not fixed by program execution. Xtensa processor loads and stores from the load/store unit are *processor ordered*, which means that writes always execute in program order but the processor may promote reads ahead of writes as long as those read operations do not bypass write operations directed to the same target address. The Xtensa LX7 processor executes read operations from the load/store unit in program order with respect to other read operations.

Consequently, memories and peripheral devices connected to the XLMI port must be designed so that they do not strictly depend on the program order of read and write operations for proper operation. However, the memories and peripheral devices attached to the XLMI bus can rely on the processor maintaining data coherence in its load/store operation ordering.

Note:

1. Xtensa processor store operations are never speculative, on any processor port including the XLMI port.
2. The processor does not fetch instructions over the XLMI port. The processor uses the XLMI port only for data operations.
3. For reference of the XLMI `RetireFlush` signal see Section 20.3.10.

20.1 Proper System Design Using the XLMI Port

The XLMI port, unlike the Xtensa RAM interface ports, incorporates signals that indoctrinate the fact—which XLMI loads were real and which were merely speculative. See the descriptions of the XLMI port's `Load`, `LoadRetire`, and `RetireFlush` signals in

Section 20.3. These three signals allow you to properly couple FIFO memories, I/O devices, and other system devices that are sensitive to speculative-read side effects to the XLMI port.

For example, FIFO memories have read side effects. If the processor takes a data word from a FIFO memory, that word no longer resides in the FIFO. However, if that load instruction is subsequently killed in the processor's pipeline, the data word fetched from the FIFO could be lost. Consequently, FIFO memories must be connected to the XLMI port through a buffer that stores the last word fetched from the FIFO until the processor signals that the data has been consumed.

Further, peripheral I/O registers such as certain status registers often initiate operations in the peripheral device when the processor reads the register. One example of such a register is a counter that increments every time a read operation to that register occurs. A better design would be a counter that increments when the processor writes to the register (instead of reading it) because the Xtensa processor's store operations are never speculative.

To reiterate: logic, external to the Xtensa LX7 processor, must be employed to couple FIFO memories, I/O devices, and other system devices that are sensitive to speculative-read side effects to prevent such side effects from adversely affecting system operations.

Another difference between the XLMI port and the RAM interface port is that the processor drives the full 32-bit address of XLMI port loads and stores. (All data loads appear on the XLMI port, whether or not they are targeted at the XLMI port.) Furthermore, for load operations, the upper portion of the load address is available early, towards the end of the E-stage (see Section 20.3.2). This characteristic can be useful when connecting shared memories and devices on the XLMI port through an arbiter that manages memory (or device) accesses initiated by multiple processors or RTL blocks. The arbiter can use the early arrival of the upper portion of the address to decode whether the load operation address is meant for the XLMI port.

Due to the complexity of designing with peripheral devices that have read side effects, Cadence recommends the following:

1. Use the processor's RAM interface ports for design situations where speculative loads will not cause adverse system effects (usually for memories, which are unaffected by the additional read traffic).
2. Use the processor's PIF interface bus to connect system components that cannot tolerate speculative loads and can tolerate the PIF's longer latency.
3. If the lowest possible latency and highest possible I/O bandwidth are required, use the processor's XLMI interface but be certain to use the XLMI port's speculative-load indicator signals to prevent adverse system effects. Devices sensitive to read side effects will not operate properly if these signals are ignored.

20.2 XLMI Port Design Considerations

The Xtensa processor consumes (loads) data from the XLMI port one cycle (for processors with 5-stage pipelines) or two cycles (for processors with 7-stage pipelines) after the processor asserts the XLMI port's address and control lines. The processor provides (stores) data to the XLMI port during the same cycle that it asserts the port's address and control lines. XLMI port reads and writes are performed in a pipelined manner with a maximum throughput of one read or write operation per cycle. Slower external devices attached to the XLMI port can assert a busy signal to postpone loads and stores.

If a designer connects a memory with read side-effects—such as a FIFO memory—to the XLMI port interface, additional external logic is required to ensure that read data is not lost until the XLMI port indicates that the processor has retired the load. The XLMI port interface eases the design of this logic by signaling when loads are initiated and retired.

This feature distinguishes the XLMI port from the processor's other local-memory interfaces because it allows for the proper design of attached devices that may have read side effects, such as FIFO memories. The XLMI port signals the initiation of loads and the retiring of loads. This feature helps SOC designers take into account the *speculative* nature of reads directed at the XLMI port.

Xtensa loads are *speculative* because a read operation on the XLMI interface-port signals does not necessarily mean that the processor will consume the read data and retire the load instruction. For performance reasons, many loads will attempt to read data from the XLMI port even though the actual load target address is assigned to another memory interface. Also, a variety of internal events and exceptions can cause pipeline flushes that subsequently cause the processor to replay loads targeted at addresses assigned to the XLMI port because the data obtained from the first load has been flushed. Consequently, all devices attached to the XLMI port must be designed to accommodate the speculative nature of all processor read operations. Appendix B provides more details on connecting memory arrays to the Xtensa processor's local-memory and XLMI ports.

The XLMI port's nominal access latency is the same as that configured for all the local memories: one cycle for a 5-stage pipeline and two cycles for a 7-stage pipeline. These latencies mean that read data will be returned to Xtensa LX7 one or two cycles after the processor drives the port's address and control lines, for 5- and 7-stage pipelines, respectively. For writes however, the XLMI port must accept store data along with the write enable and it must update memory contents within one cycle, regardless of the statically configured memory-access latency. In other words, the latency between a data write and the earliest read that sees the new data cannot exceed one cycle regardless of pipeline configuration. Access latency and same-address-write-to-read latency is the same across all local Xtensa LX7 memory ports including the XLMI.

20.2.1 XLMI Port Options

Xtensa processor cores offer the following XLMI port options:

- Mapped to region of size: 0.5, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 Kbytes or 1, 2, or 4 Mbytes
- Multi-master arbitration for devices attached to the XLMI port (through use of the `Busy` signal)

Another Xtensa processor option that affects the XLMI port is:

- The XLMI port's access width is the same as the configured processor's data-memory ports. Width options are 32, 64, or 128 bits.

Note: There are some restrictions on the XLMI port versus other local-memory ports:

- The `S32C1I` will not operate over the XLMI port
- XLMI port addresses are always identity mapped, even with the full TLB MMU option.
- Inbound-PIF requests to the XLMI port are not allowed and will generate a bus error.
- Dual load/store units are not allowed

20.3 XLMI Signal Descriptions

The XLMI port is also referred to as the "data port" to distinguish it from the other Xtensa processor local-memory ports and that term is used for the XLMI port's signal-naming conventions to underscore the XLMI port's ability to deal with speculative loads.

Table 10–24 lists the signals for an interface to data port *n* from load/store unit *m* The signal-naming convention includes the memory name and number and the load/store unit number. For example, `DPort0Addr0` are the address signals to `DataPort0` driven by load/store unit 0.

Note:

- Xtensa processors can have a maximum of one XLMI port, so *n* is always 0.
- Xtensa processors cannot configure XLMI and dual load/store units, so *m* is always 0.

See Section 35.9 on page 633 for information about XLMI AC timing. The following sections provide details about the XLMI signals (per load/store unit).

20.3.1 XLMI Port Enable

This is the primary enable used to activate device(s) connected to the XLMI port. This signal indicates a request for a read or write transaction on the port. The device connected to the XLMI port is free to reject the transaction by asserting DPort*n*Busy*m* on the next cycle if the busy option has been configured. Similarly, the processor is free to abort the transaction by negating DPort*n*Load*m* on the next cycle or by asserting DPort*n*RetireFlush*m* in the correct order. Transaction details are given in Section 20.4 “XLMI Transaction Behavior” on page 378.

20.3.2 XLMI Port Address

The XLMI port is mapped to a contiguous address range whose start address must be an integer multiple of its size. Any number of the types of devices listed in the overview above can be connected to this interface port as long as they are mapped within the specified region. The address decoding for these devices and the multiplexing of write or read data must be performed by logic external to the processor core.

The XLMI port must be mapped to a range whose start address is an integer multiple of the defined size of the port’s address space. The port address is indexed by only some portion of the virtual address that depends on size and access width. Specifically, that portion is [log2(size_in_bytes)-1:log2(access_width_in_bytes)].

Even though the XLMI port’s address is indexed, the processor drives the full 32-bit virtual address of XLMI port loads and stores. The upper portion of this 32-bit address can be useful in external address decoding, and is available a pipeline stage earlier than the "hit" signal DPort*n*Load*m*.

The lower portion of the XLMI’s address output is reserved for future use, is always undefined, and should presently be ignored by external logic. Note that this behavior differs from other Xtensa local-memory ports where the lower address bits are not output—that is, the address output of a local memory port is always aligned to its access width. For example, if an instruction RAM’s access width is 32 bits, the address output by the Xtensa processor on that instruction-RAM port does not include bits [1:0], because IRam0Addr is only significant for bits 2 and above.

To reiterate, the processor does output the least-significant address bits for the XLMI port, but the meaning of these bits are undefined under all circumstances. It is not possible to derive any information about the actual load/store address based on these bits.

For example, DPort0Addr0 is only significant from bit 3 on up for a 64-bit-wide XLMI port. The address bits representing unaligned bytes within the access width are never meaningful. Even if the load or store operand size is less than 64 bits, and that operand is not 8-byte aligned (that is, aligned to the 64-bit access width), XLMI address bits [2:0]

will be undefined. For a load, the processor expects to receive a 64-bit quantity from which it extracts (by way of alignment) the bytes appropriate for the load. For a store, the appropriately aligned data is output as a 64-bit transaction.

20.3.3 XLMI Byte Enables

XLMI byte enables are valid for both loads and stores. For example, for a 64-bit XLMI port, even though the 64-bit quantity at `DPort0Addr0[31:3]` is expected during a load, the processor will only use the bytes indicated by `DPort0ByteEn0[7:0]`.

And even though the processor outputs a 64-bit quantity during a store, only the bytes indicated by `DPort0ByteEn0[7:0]` should be written into the memory location pointed to by `DPort0Addr0[31:3]`.

Because the TIE language offers an arbitrary-byte-disable feature, the XLMI port's byte enables do not necessarily indicate the size or alignment of the load or store operation. Instructions created with TIE can assert any arbitrary combination of byte enables on the XLMI port. Therefore, the XLMI port's byte enables only indicate the active byte lanes for a data transaction and they may not indicate the data's size or alignment.

Devices connected to the XLMI port that must work correctly under any conditions, including conditions where an Xtensa processor configuration uses designer-defined instructions with arbitrary byte enables, must be designed to send or receive data only on the designated byte lanes and must not infer operand size or alignment from any combination of asserted byte enables. Devices designed for processor configurations that do not employ arbitrary byte enables can be somewhat simpler.

20.3.4 XLMI Write Enable

Indicates a write transaction. `DPort`*`n`*`Wr`*`m`* is single bit.

20.3.5 XLMI Write Data

XLMI write data bus from the processor. This bus has the same width as the read data bus `DPort`*`n`*`Data`*`m`*. Only the bytes on the byte lanes indicated by the `DPort`*`n`*`-ByteEn0`*`m`* signals are valid.

20.3.6 XLMI Busy

Arbiters for XLMI-connected devices that are shared by multiple processors or other sorts of masters can use the XLMI `Busy` signal to assist in access arbitration for the device. This signal indicates that the resource isn't immediately available. The `Busy` configuration option must be chosen for the XLMI port to make this signal available. Note

that the Busy signal terminates an XLMI transaction. The processor may elect to retry the transaction on the next cycle, it may wait and retry the transaction later (possibly after other intervening XLMI transactions occur), or it may abandon the transaction and never retry it.

As with all the other local memory port signals, there is one XLMI Busy signal associated with the load/store element's XLMI port interface. Assertion of this signal when a load is attempted from the XLMI port will usually stall the pipeline. Assertion of this signal when a store is attempted to the XLMI port may cause a pipeline stall depending on the state of the associated load/store unit.

The pipeline stall is a global stall that causes all stages of the processor pipeline to stall. In the case of global stall due to a Busy response to a load, this means that the instruction immediately older than the load can be held up. In the 7-stage pipeline version of the Xtensa LX7 processor, this potentially creates a deadlock situation if the older instruction needs to retire for the Busy assertion to be cancelled. If this situation can arise in a system, it must have a mechanism to terminate Busy assertion after a timeout period.

The processor samples the XLMI port's Busy interface signal one cycle after the processor drives the address and enable lines, regardless of the statically configured memory latency (determined by the pipeline length of one or two cycles). For loads, this point is always during or before the cycle that data is normally expected back from the addressed device.

When Busy is asserted, the processor may choose to retry the load or store, or it may choose to try a brand new load or store. In other words, Busy terminates a transaction, it does not extend the transaction. The Xtensa processor may or may not restart the terminated transaction at a later time.

Note: Assertion of the XLMI port's Busy interface signal in cycle n always applies to the XLMI transaction begun in cycle n-1. Device(s) connected to the XLMI port must base their assertion of Busy in cycle n on the registered state of the XLMI's address and control ports in cycle n-1. Xtensa processors use the value of Busy in cycle n to decide what values to place on the XLMI port's address and control outputs *during cycle n*. In other words, there exist combinational-logic paths within the XLMI port's logic from its Busy input to *all* of the XLMI port's outputs including En, Addr, ByteEn, Wr, WrData, Load, LoadRetired, and RetireFlush. Under no circumstances should external logic assert an XLMI port's Busy input signal during cycle n based on the state of the XLMI's address or control outputs during that same cycle. Doing so leads to a combinational-only logic loop, which creates an inappropriate feedback path. It is up to the system designer to ensure that implementation and timing of external circuits attached to the XLMI port meet these requirements to ensure the proper synchronous behavior of the port.

20.3.7 XLMI Load Signal

The Xtensa processor decodes load/store "hits" for the contiguous XLMI port region. This means checking whether the physical address of the load/store instruction (which is the same as the virtual address calculated in the E stage) falls within the statically configured XLMI port region. The "hit" signal for loads is internally qualified by a number of conditions to determine whether the processor really wants to initiate a transaction on the XLMI port. If it does indeed choose to do so, the processor asserts DPort*n*Load*m* to indicate to the XLMI-attached device that a load transaction has begun.

The DPort*n*Load*m* signal is gated by the Busy signal from the XLMI port.

20.3.8 XLMI Data Bus

After the device attached to the XLMI port recognizes that a load transaction has begun, it must return data to the processor. Data must be returned zero or one cycle after assertion of the Load signal, depending on the configured local memory access latency (zero cycles for a 5-stage pipeline and one cycle for a 7-stage pipeline).

The XLMI read data bus has the same width as the write data bus DPort*n*WrData*m*.

20.3.9 XLMI Load Retired

After a load transaction on the XLMI port has begun, the attached device returns data as described above. But this step does not conclude the transaction because there is no guarantee that the Xtensa processor will use the data due to the speculative nature of speculative loads. At some arbitrary point after the transaction has begun, the processor asserts DPort*n*LoadRetired*m* to indicate that it did indeed consume the data. This signal tells the XLMI-attached device that it no longer needs to maintain the supplied data in a restorable state. Note that this signal need only be used by devices attached to the XLMI port that have read side effects.

Simple memories such as RAMs and ROMs do not have read side effects. The processor can read the same location many times without affecting the contents of the addressed memory location. FIFO buffers and most I/O devices do have load side effects: after a piece of data is read, it's gone. Such devices must be designed to save the contents of the most recent read until DPort*n*LoadRetired*m* indicates that the processor has retired the load.

Note that during the entire transaction—that is, between transaction initiation and assertion of DPort*n*LoadRetired*m*—the Xtensa processor could initiate additional load transactions. Therefore, each assertion of DPort*n*LoadRetired*m* assertion applies

only to the oldest outstanding load transaction on the XLMI port from the load/store unit. Unless a flush is indicated (see Section 20.3.10), there will be separate assertions of DPort*n*LoadRetired*m* for every XLMI load transaction.

20.3.10 XLMI Retire Flush

As described above, a load transaction on the XLMI port completes some arbitrary number of cycles after it has begun, during which additional load transactions can be initiated. There are some situations—mainly due to a pipeline flush and replay—when the processor must abandon all outstanding transactions. The processor signals this condition by asserting DPort*n*RetireFlush*m*.

An XLMI load transaction can end in only one of two ways. Either the Xtensa processor asserts DPort*n*LoadRetired*m* to indicate successful completion, or it asserts DPortnRetireFlushm to indicate cancellation. The difference between these signals is that DPort*n*LoadRetired*m* applies to only one outstanding XLMI load transaction and DPortnRetireFlush*m* applies to all outstanding XLMI transactions.

The maximum number of load transactions that can be outstanding at any given time for each load/store unit is equal to one plus the local-memory access latency, which is two outstanding transactions for a processor configuration with a 5-stage pipeline and three outstanding transactions for processor configurations with a 7-stage pipeline.

20.4 XLMI Transaction Behavior

The basic transactions common to all local memories attached to an Xtensa processor are described in Appendix B. It may be helpful to view that section before reviewing the XLMI transactions discussed in this section, because the memory transactions are similar and simpler.

20.4.1 General Notes

A store-to-load bypass can occur for transactions directed at addresses assigned to the XLMI port, which means that a load from an XLMI address that immediately follows a store transaction to the same address may get the return value from the processor's internal store buffer (where the stored value may be residing prior to the store transaction actually occurring) instead of from the device attached to the XLMI bus. This bypass feature improves processor performance when dealing with simple memories, but such a bypass transaction is potentially a problem for transaction targets with volatile contents, such as I/O device registers (which can be changed by events and other devices external to the Xtensa processor) and memory resources shared by multiple processors, including dual-ported RAMs (whose contents can be changed by the "other" processor). The Xtensa processor's MEMW instruction can be used to prevent this problem by forcing

all pending XLMI load and store transactions to complete before any such operations following the MEMW instruction are performed. Also note that the MEMW instruction's use is not limited to XLMI transactions.

20.4.2 Busy

Figure 20–100 shows a Busy assertion for an XLMI load for an processor with a 5-stage pipeline. If Busy is asserted in cycle 4, the processor would probably choose to retry the load. If it so chooses, the output-signal states required for performing the load are repeated, precluding the need for any signal latching by external logic. The Xtensa processor determines in cycle 4 whether to retry the transaction or proceed to another transaction based on the value of DPort0Busy0 *in cycle 4*. In other words, Xtensa processors have a combinational-logic-only path between the DPort0Busy0 signal and the XLMI port's address and control signals to repeat these output states when the Busy signal is asserted.

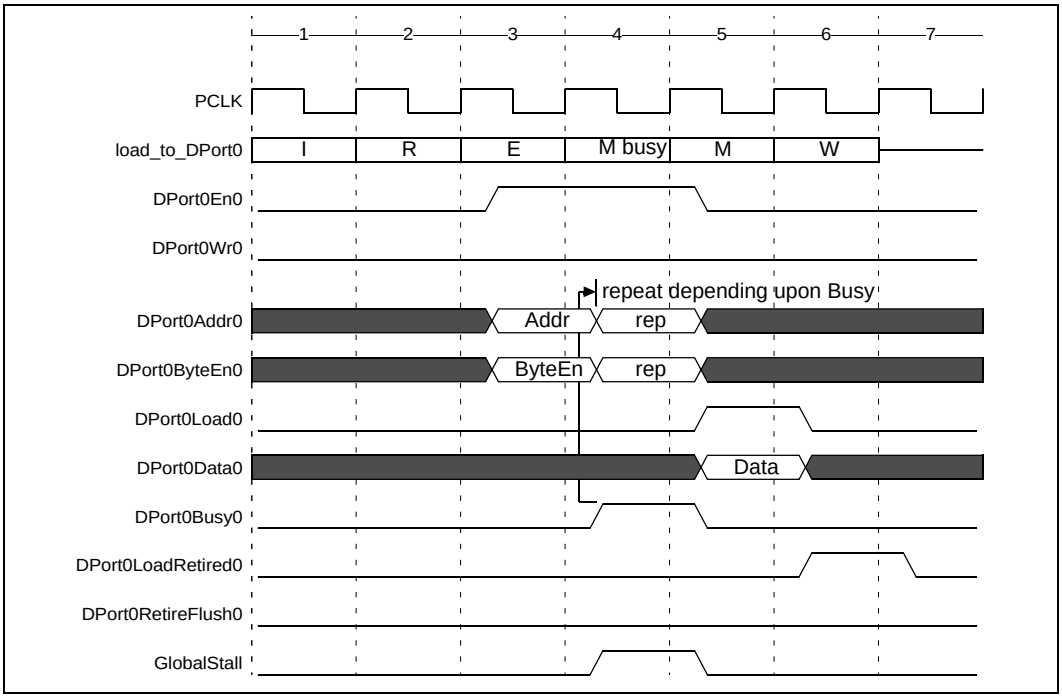


Figure 20–100. XLMI Load with Busy (5-Stage Pipeline)

To retry the load, the processor must stall its pipeline. This stall is shown in Figure 20–100 by the "M busy" label in cycle 4 and by the assertion of the internal signal GlobalStall. Assertion of GlobalStall in a cycle indicates that the processor pipeline is stalled during that cycle.

Note:

The `GlobalStall` signal shown in Figure 20–100 is not an external XLMI signal. It is an internal processor signal and is only shown here to further illustrate the example. The arrow from `Busy` assertion to the address and control lines in Figure 20–100, is implied and is omitted from Figure 20–102 onwards.

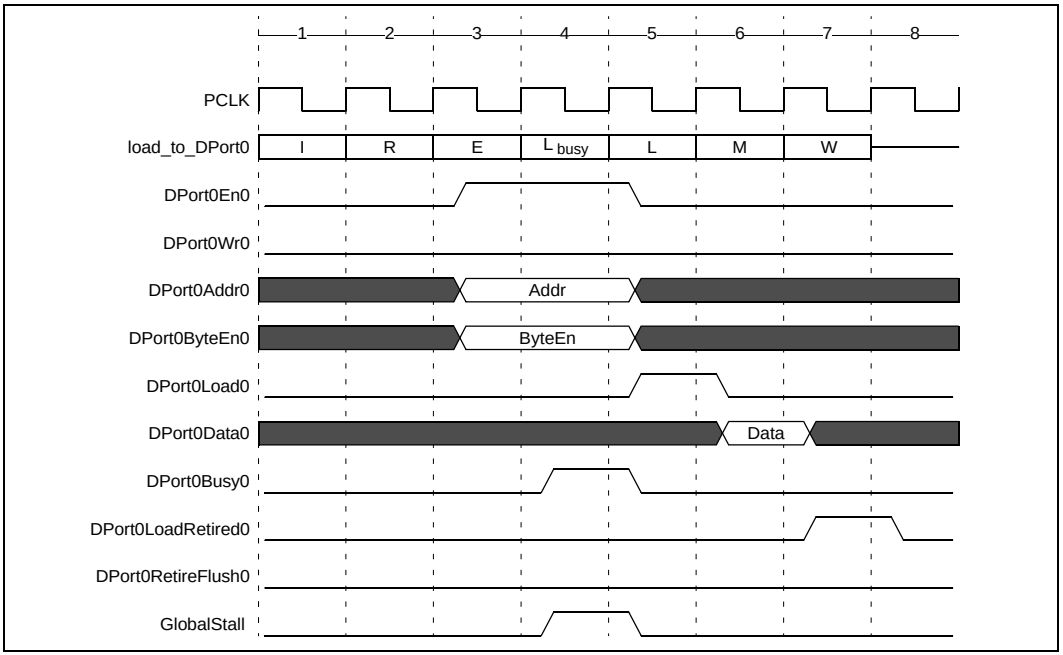


Figure 20–101. XLMI Load with Busy (7-Stage Pipeline)

Figure 20–101 shows the XLMI load-busy assertion case for a processor with a 7-stage pipeline. In this case too, the processor chooses to retry the load after seeing the assertion of `Busy`.

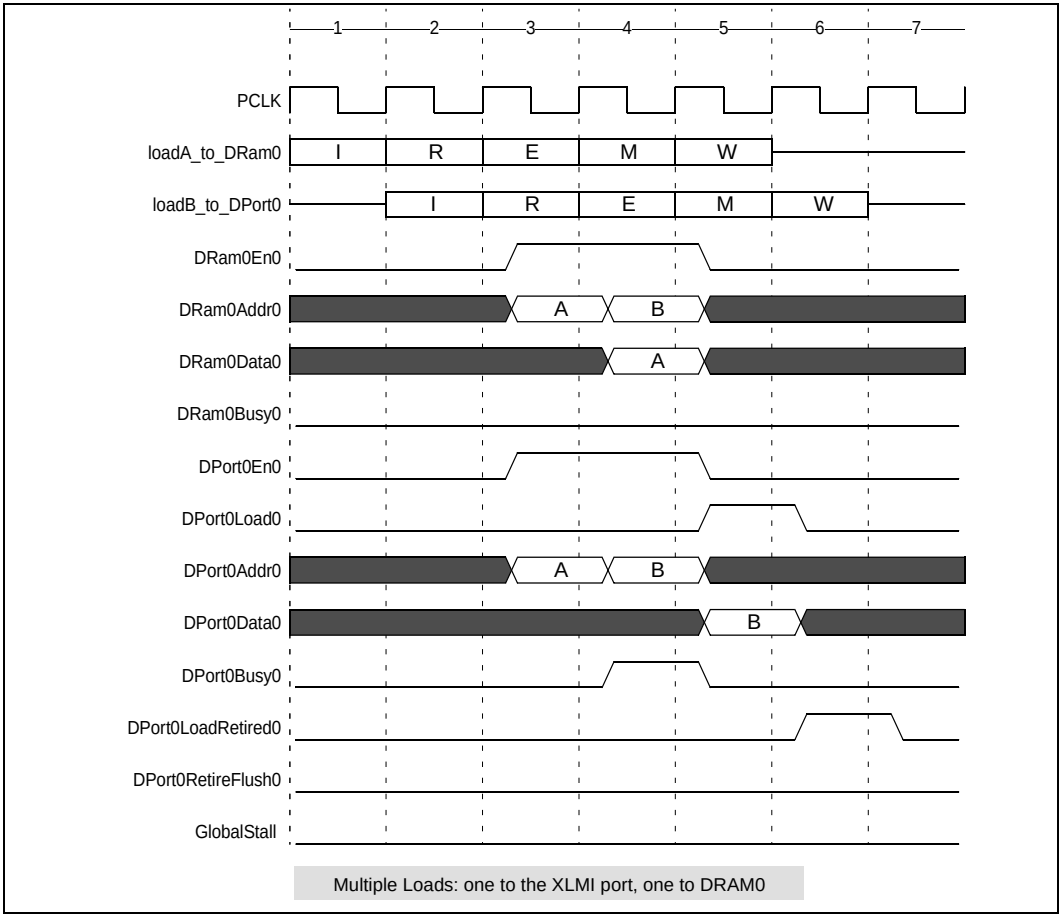


Figure 20–102. Back-to-Back Loads with Busy (5-Stage Pipeline)

Figure 20–102 shows the operation of the XLMI busy signal when there are back-to-back loads. Several points are demonstrated in the figure:

- No stall occurs during the `DRam0` load—in cycle 4—even though the `Busy` input to the XLMI port is asserted. Restating for the general case, no pipeline stall occurs if the `Busy` signal of a non-target local memory is asserted.
- The processor asserts the XLMI port address and control signals—in cycle 4—even though the XLMI port's `Busy` signal is asserted during this cycle. In the general case, the address and control signals from the processor to local-memory ports and the XLMI port will be asserted even during a cycle where the port's `Busy` signal is asserted. The processor expects either the assertion of the associated `Busy` signal or data back from the memory during the next cycle.

- The XLMI port asserted its Busy signal in response to the address and enable of cycle 3. The address in the next cycle is completely different because a new transaction has begun. Restating for the general case, in the cycle that a given local memory asserts its Busy, the processor may elect to initiate a different transaction from the one that generated the Busy indication. *Assertion of address and control in one cycle followed by a Busy assertion in the next cycle constitutes a complete transaction.*

Figure 20–103 shows Busy being asserted in response to a store. In this case, the processor chooses to retry the store in the next cycle, so it again presents the address, data, and control for the store transaction.

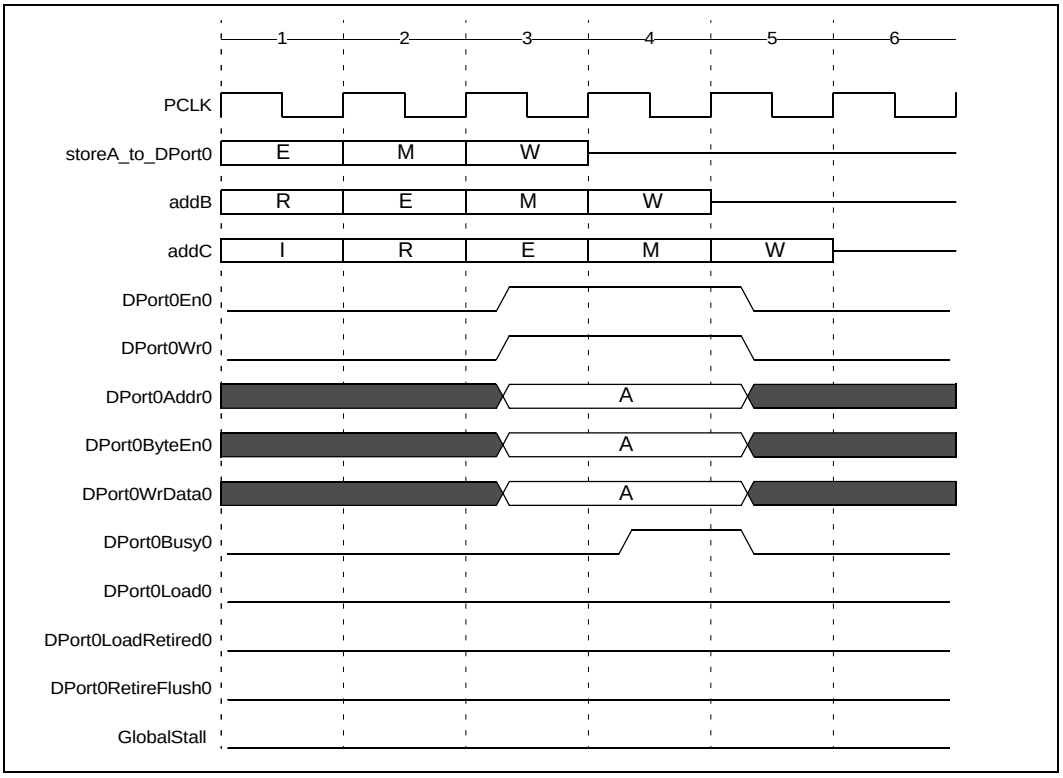


Figure 20–103. Busy to Store (5-Stage Pipeline)

Note that the internal GlobalStall signal shown in Figure 20–103 does not go high during the store operation because the operation completes after placing the data to be stored into the processor's store buffer. GlobalStall will only go high during a store operation if the processor's store buffer fills up, forcing a stall.

Figure 20–104 shows the assertion of Busy in response to a store transaction (this example is for an Xtensa LX7 configuration with a 7-stage pipeline). As discussed in Appendix B, the write timing is the same for both a 5- and 7-stage pipeline: a read transaction to the same address in the cycle following the write transaction must retrieve the newly written data.

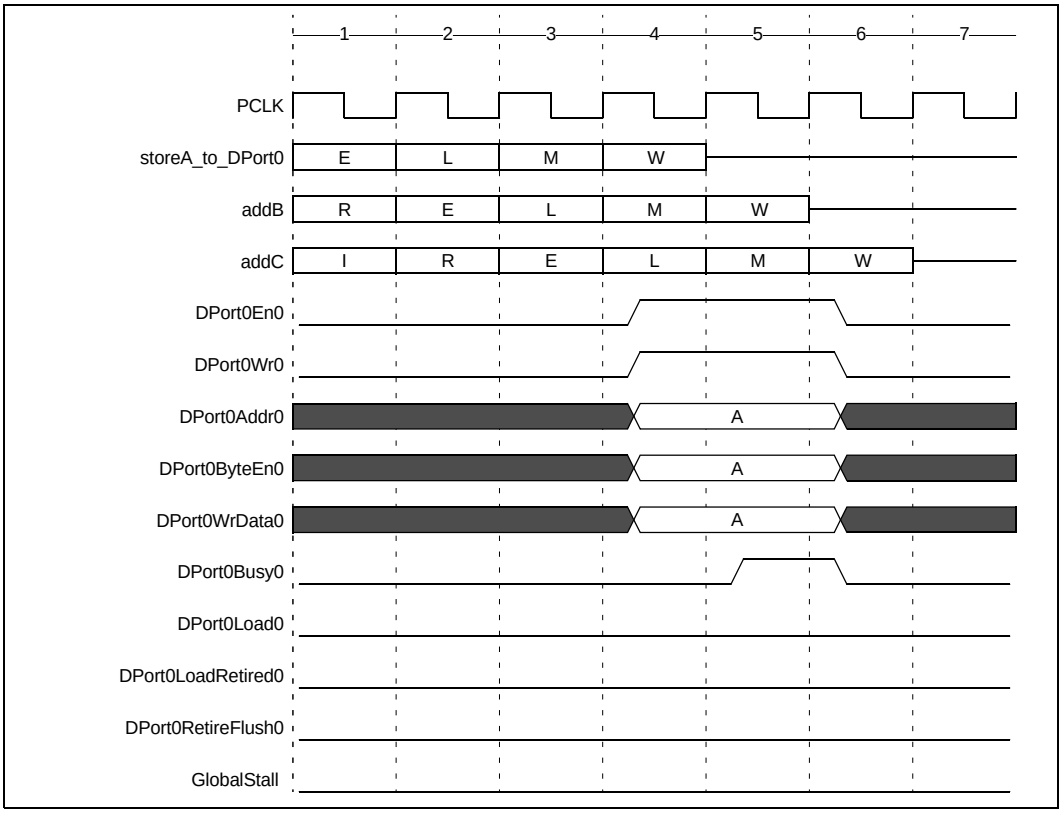


Figure 20–104. Busy to Store (7-Stage Pipeline)

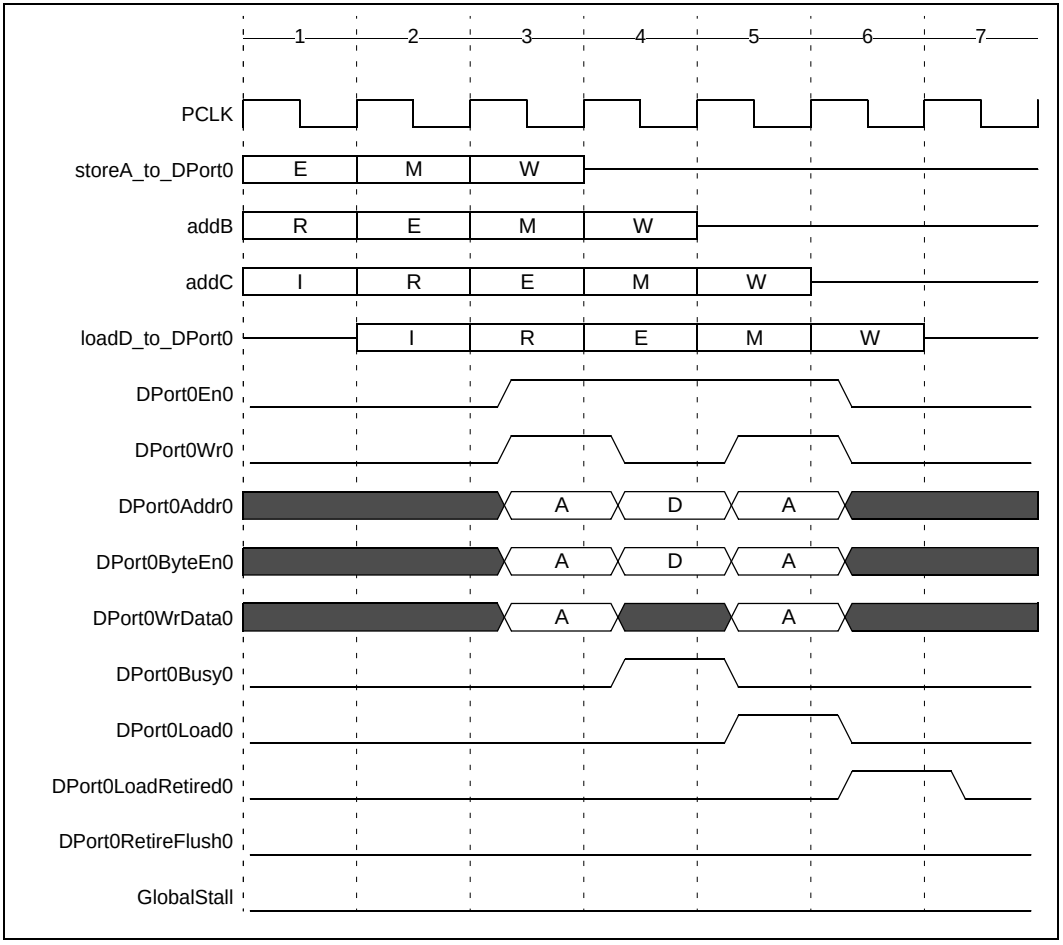


Figure 20–105. Processor Tries Different Transaction after Busy Terminates Store

Figure 20–105 shows that Busy terminates the transaction. Store A is terminated by the assertion of Busy in cycle 4. Instead of retrying the store, the processor decides to try load D. In general, local memory access is rearbitrated every cycle.

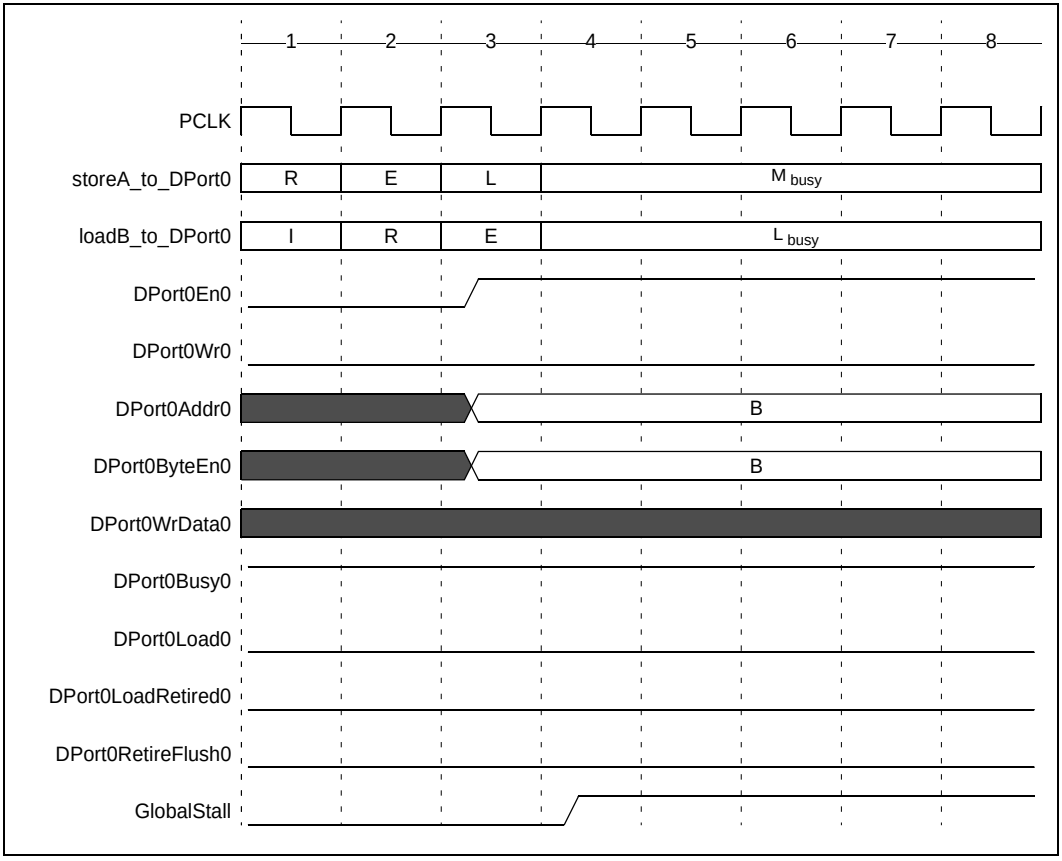


Figure 20–106. Deadlock Condition - Store to FIFO in Followed by Load from FIFO Out

Busy can cause a global stall, which causes all processor-pipeline stages to stall. If the global stall is due to the assertion of a Busy in response to a load, the instruction immediately older than the load can be held up in a 7-stage Xtensa LX7 configuration. Because loads have high priority, the Xtensa LX7 processor continuously retries the read transaction associated with the load, which can potentially cause a deadlock situation if the older instruction needs to retire for the Busy to be negated.

Consider a situation where a FIFO's input and output are attached to an Xtensa LX7 processor's XLMI port. In this configuration, an XLMI store operation writes to the FIFO's input and an XLMI load operation reads the same FIFO's output. (Although this configuration is somewhat contrived—it is not likely that a real system would be configured this way—this example makes the deadlock timing hazard very easy to see.) Busy is asserted whenever the FIFO is empty. With the FIFO initially empty, Figure 20–106 shows a store immediately followed by a load in a 7-stage Xtensa LX7 configuration. A deadlock results because assertion of Busy stalls not only the load but also the immedi-

ately preceding store. However, the Busy will not be released until the store completes (that is, reaches its W stage), which creates the deadlock. This deadlock does not occur for 5-stage Xtensa LX7 configurations and in the 7-stage Xtensa configuration, only the instruction immediately preceding the load is affected.

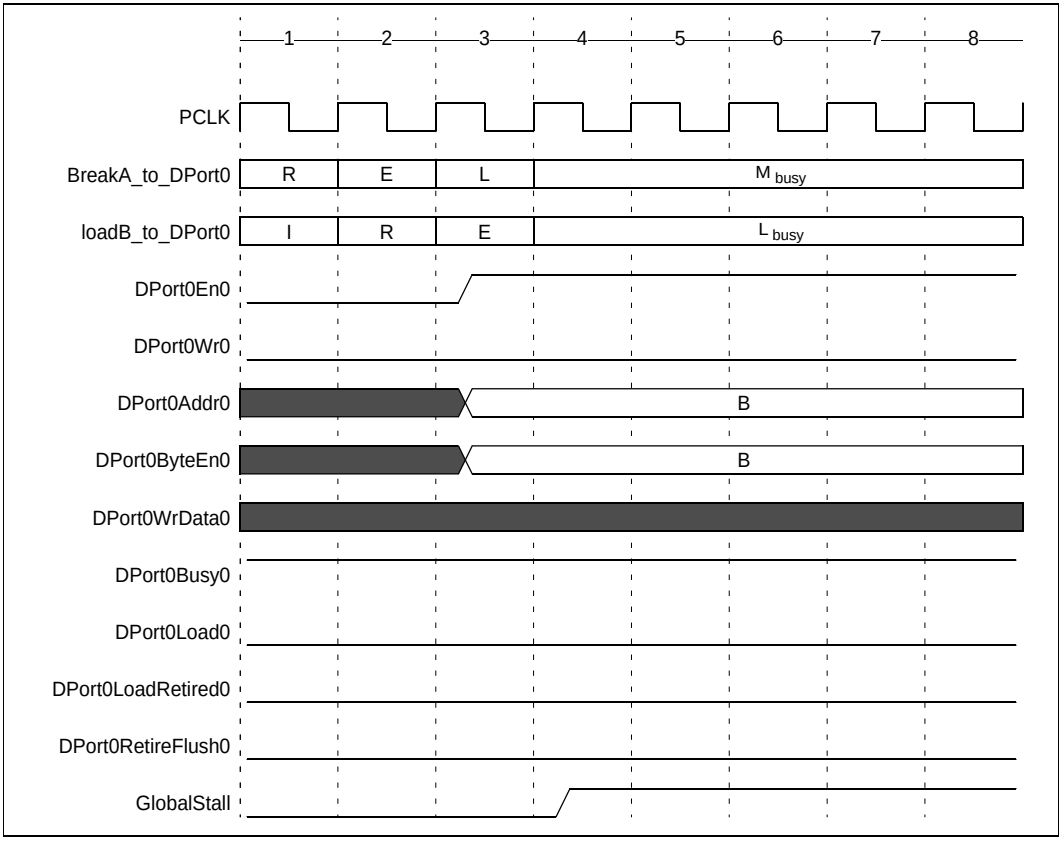


Figure 20–107. Undebuggable Processor Hang

Consider a different situation where Busy is indefinitely asserted for a reason not related to the processor's operation. Because the Xtensa LX7 processor continuously retries the read transaction associated with the load operation, there is an indefinite global stall. To debug the hung processor, the system designer places a BREAK instruction immediately prior to the load, as shown in Figure 20–107. The break exception never occurs because the instruction immediately older than the load does not retire. Again, this situation does not occur in 5-stage Xtensa LX7 configurations, and only the instruction immediately preceding the load is affected.

To avoid these situations, the system designer must ensure that there is a way to negate Busy after a timeout period.

20.4.3 Supporting XLMI Speculative Loads

The Xtensa processor's XLMI port provides special support for speculative loads (the other local memory ports do not). The XLMI port handles speculative loads through its `DPortnLoadm`, `DPortnLoadRetiredm`, and `DPortnRetireFlushm` signals. These three signals are absent from the processor's other local-memory ports (as opposed to earlier versions of Xtensa processors). Therefore, local devices with read effects (such as I/O devices) should be connected to the XLMI port and not to the processor's local-memory buses.

The `DPortnLoadm` signal is asserted the cycle following an XLMI enable (for both 5- and 7-stage pipelines) to indicate that the load target is the XLMI port. This sequence is shown in Figure 20–108. Operation of the `DPortnLoadm` signal is partly based on the state of the associated `Busy` signal from the device attached to the XLMI port. `DPortnLoadm` will not be asserted if the XLMI request receives a `Busy` response. If the processor does not assert the `Load` signal during the cycle following the `Enable`, no load transaction over the XLMI port has begun from the processor's perspective.

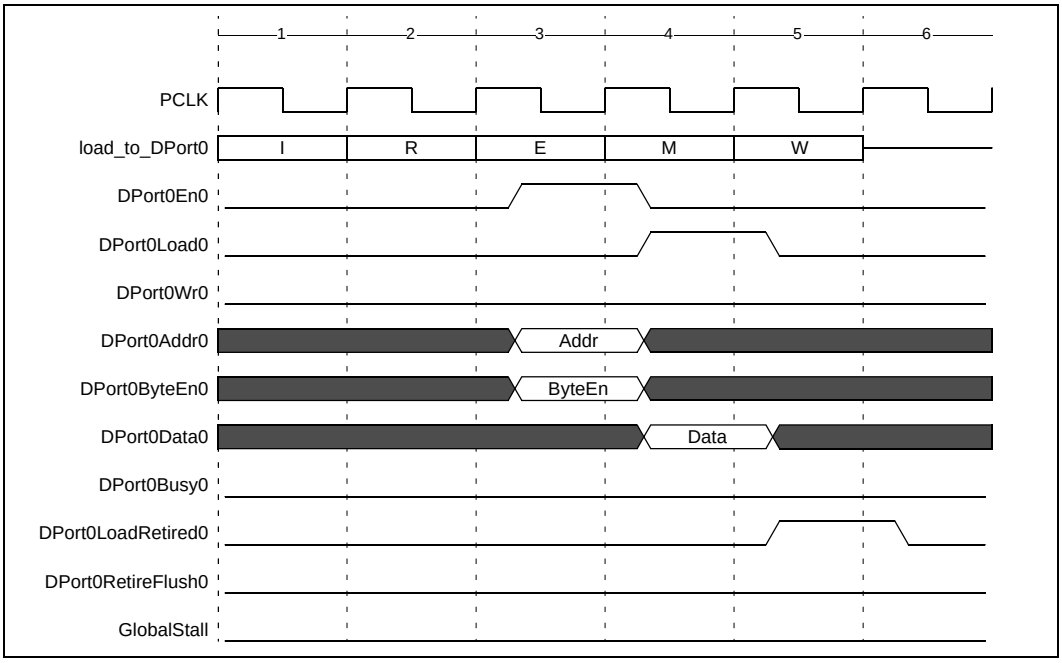


Figure 20–108. Load and Retired Assertion (5-Stage Pipeline)

Assertion of `LoadRetired` by the processor means that the oldest outstanding XLMI port load from the load/store unit has retired. At the earliest, `DPortnLoadRetiredm` is asserted *local_memory_access_latency* cycles after `DPortnLoadm`. `DPortnLoadRetiredm` will not be asserted if there are no outstanding XLMI loads.

Assertion of RetireFlush means that all outstanding XLMI loads have failed to retire. The processor will not assert both LoadRetired and RetireFlush during the same cycle.

Devices with load side effects that are attached to the XLMI port must observe the XLMI port's DPortnLoadRetiredm and DPortnRetireFlushm signals. These devices must be designed so that they undo the side effects of any unretired loads.

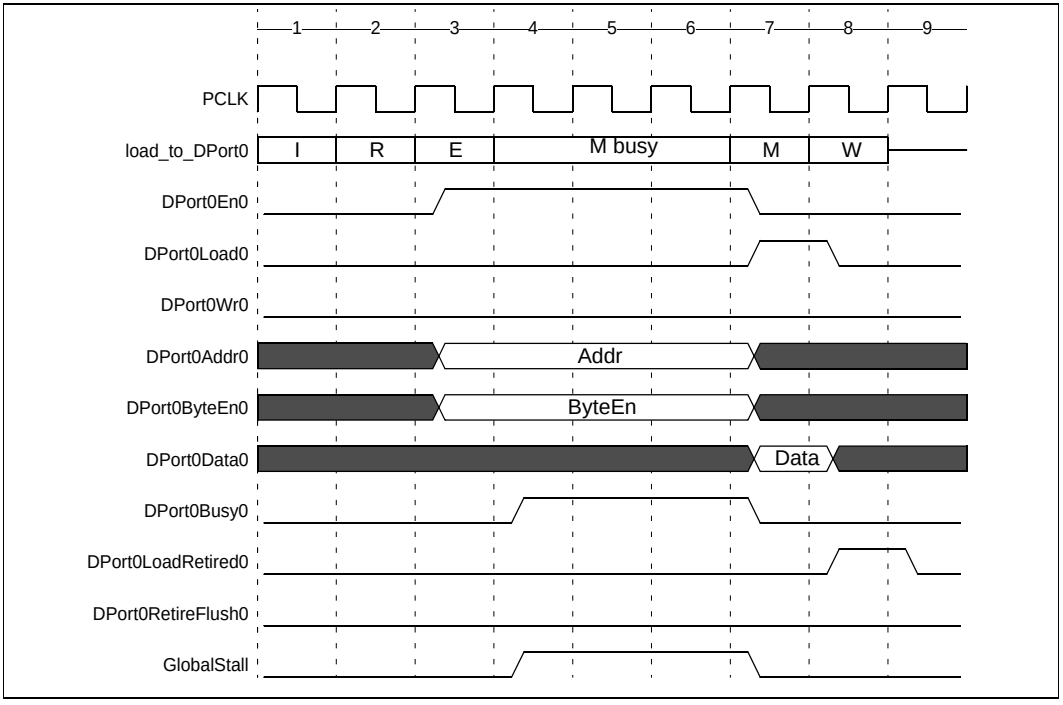


Figure 20–109. Busy Restarts XLMI Transaction (5-Stage Pipeline)

Figure 20–109 shows that if the device attached to the XLMI port asserts Busy, the transaction starts anew, and the attached device must re-examine the DPort0Load0 signal to decide whether or not to wait for the DPort0LoadRetired0 or DPort0RetireFlush0 signals. In other words, Busy definitively ends the transaction and there is only one Load assertion for a given LoadRetired. The load transactions initiated in cycles 3, 4, and 5 are terminated by the assertion of Busy in cycles 4, 5, and 6. The load transaction initiated in cycle 6 proceeds to successful completion.

Figure 20–110 shows the same as the above for a 7-stage pipeline.

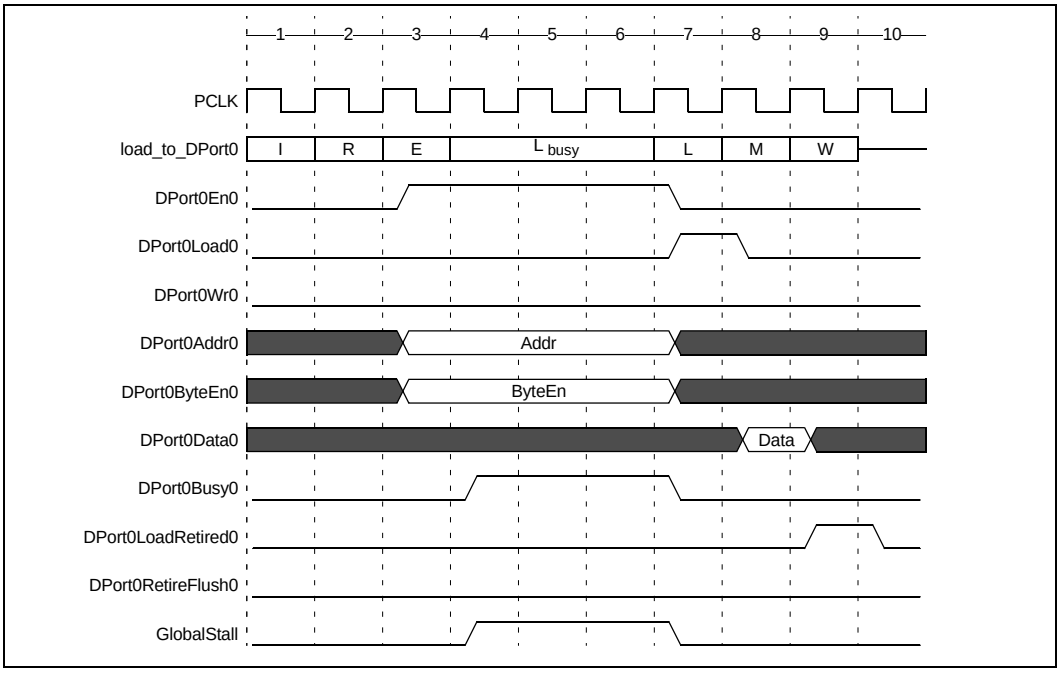


Figure 20–110. Busy Restarts XLMI Transaction (7-Stage Pipeline)

The following figure shows that the XLMI port signals that support speculative loads operate independently of pipeline stalls. `Load` is always asserted one cycle after the enable, whether or not `GlobalStall` is also asserted in the cycle. Similarly, as long as `DPort0Busy0` is not asserted, data is accepted from the XLMI port whether or not `GlobalStall` is asserted in the cycle. As soon as the processor determines that the load instruction will commit, it asserts `DPort0LoadRetired0`, whether or not `GlobalStall` is asserted. If the processor instead kills the load instruction, it asserts `DPort0RetireFlush0`, whether or not `GlobalStall` is asserted. In this figure, the assertion of
`DPort0LoadRetired0` applies to and terminates the earliest load (started in cycle 3). The assertion of `DPort0RetireFlush0` terminates the second load (started in cycle 7).

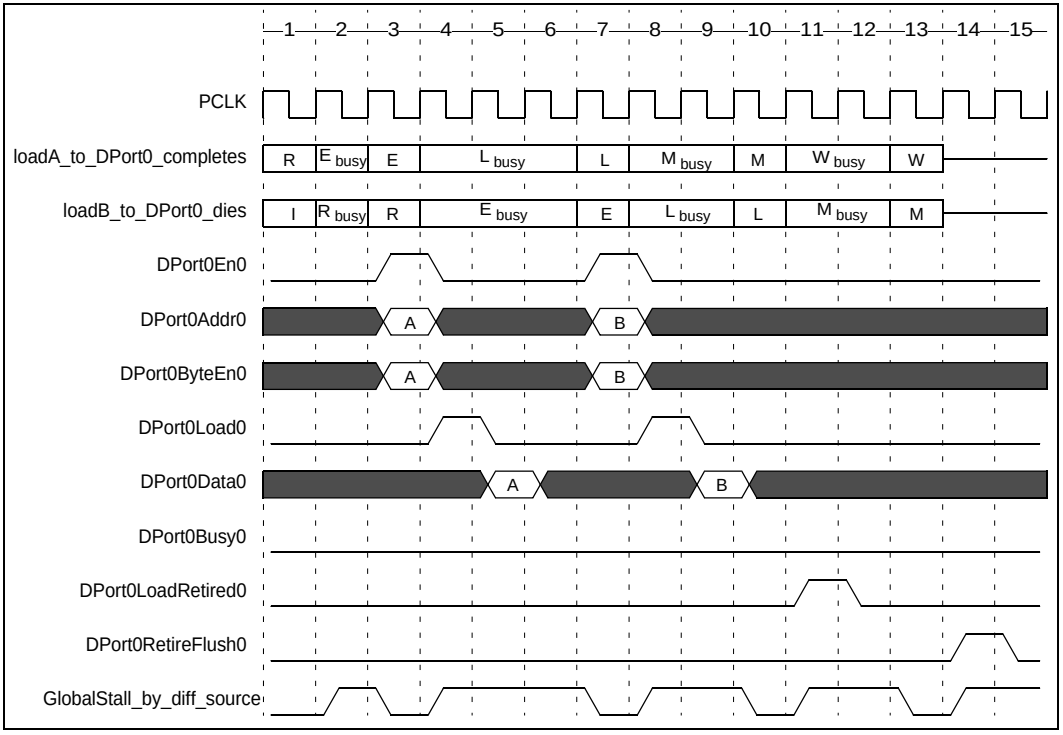


Figure 20–111. XLMI Speculative Load Support Signals Independent of Stall

Figure 20–112 and Figure 20–113 show that there can be a maximum of two outstanding load transactions for a 5-stage pipeline and three outstanding load transactions for a 7-stage pipeline. Therefore, a device with read side effects must be able to undo the effects of double the number of loads when attached to Xtensa configurations with multiple load/store units.

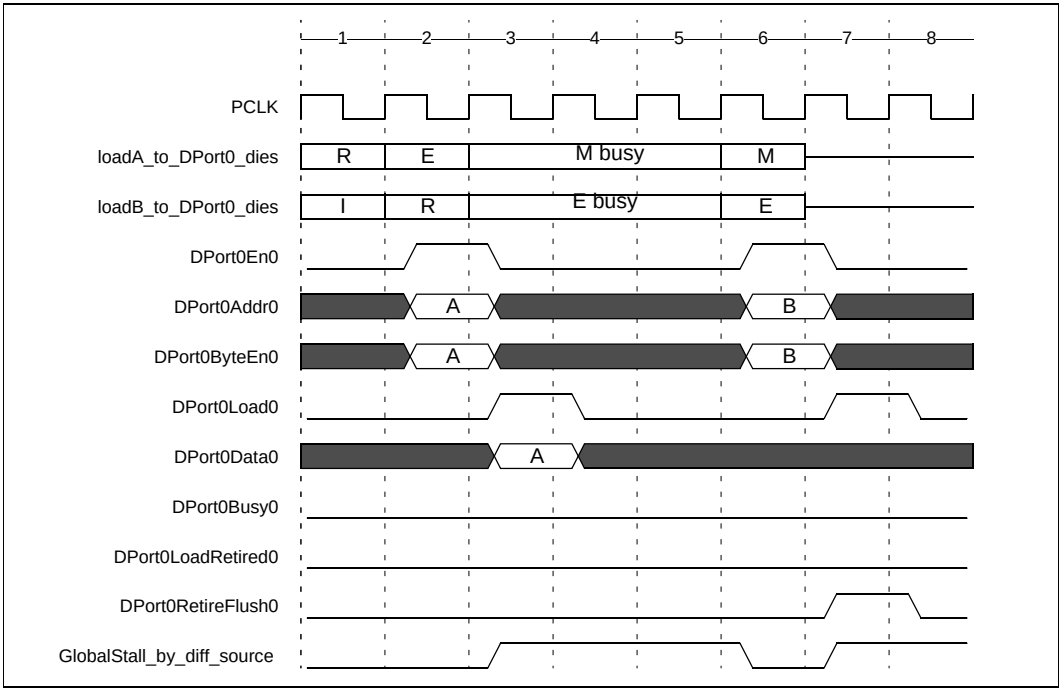


Figure 20–112. Maximum of Two Outstanding Loads (5-Stage Pipeline)

Figure 20–112 also demonstrates an important characteristic of the `Load` signal that the XLMI device designer must consider. The `Load` signal for the last load in this figure is asserted (in cycle 7), even though `RetireFlush` is also asserted in the same cycle. In this case, assertion of `Load` does not signify the start of a new load transaction because `RetireFlush` is also asserted in the same cycle.

Note: `RetireFlush` nullifies all outstanding loads, even if the `Load` signal is asserted concurrently with the assertion of the `RetireFlush` signal. Consequently, XLMI devices must always qualify `Load` with `RetireFlush`.

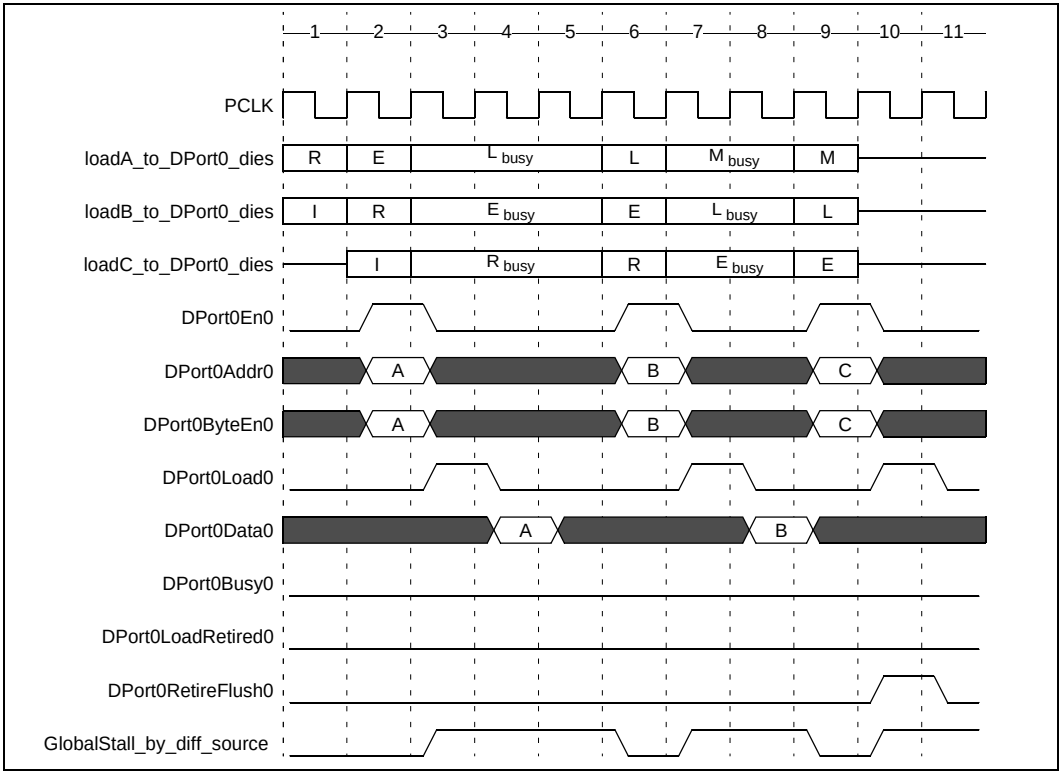


Figure 20–113. Maximum of Three Outstanding Loads (7-Stage Pipeline)

Figure 20–113 also demonstrates this important characteristic of the `Load` signal shown in Figure 20–112. The `Load` signal for the last load in is asserted in cycle 10, even though `RetireFlush` is also asserted in the same cycle. In this case, assertion of `Load` does not signify the start of a new load transaction because `RetireFlush` is also asserted in the same cycle.

Note: `RetireFlush` nullifies all outstanding loads, even if the `Load` signal is asserted concurrently with the assertion of the `RetireFlush` signal. Consequently, XLMI devices must always qualify `Load` with `RetireFlush`.

20.4.4 Stores and Support for Speculative Transactions

Stores are never speculative and therefore require no special support. Figure 20–114 shows store operations for the Xtensa LX7 processor.

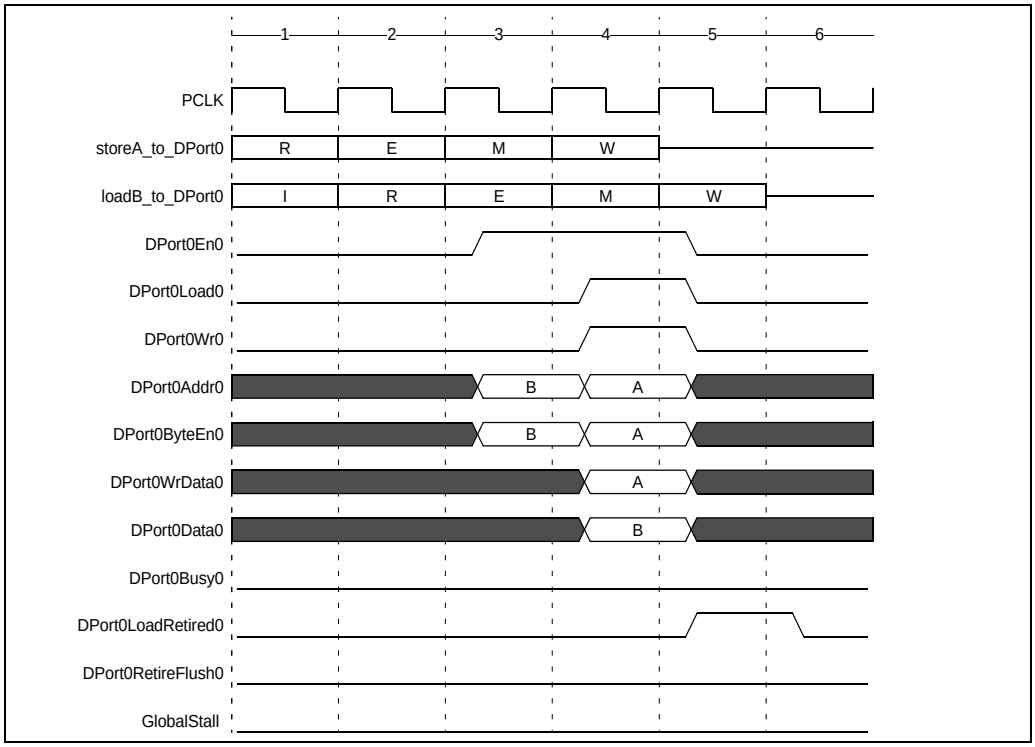


Figure 20–114. XLMI Store (5-Stage Pipeline)

Figure 20–115 show stores for 7-stage pipelines for the Xtensa LX7 processor.

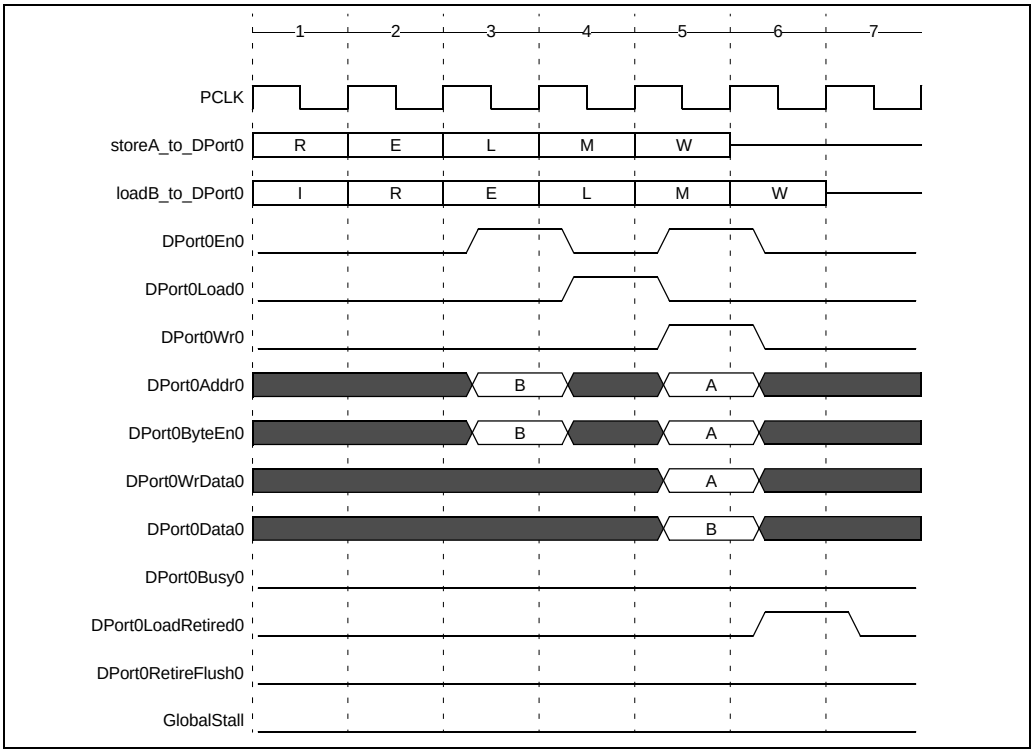


Figure 20–115. XLMI Store (7-Stage Pipeline)

Because stores and loads are independent, writes can occur during a variety of speculative-load scenarios. Writes are never speculative and should not be flushed.

Figure 20–116 shows a write transaction initiated in the same cycle that `LoadRetired` is signalled. The write was delayed one cycle past the W stage of the store for an unrelated cause, such as the presence of older queued stores.

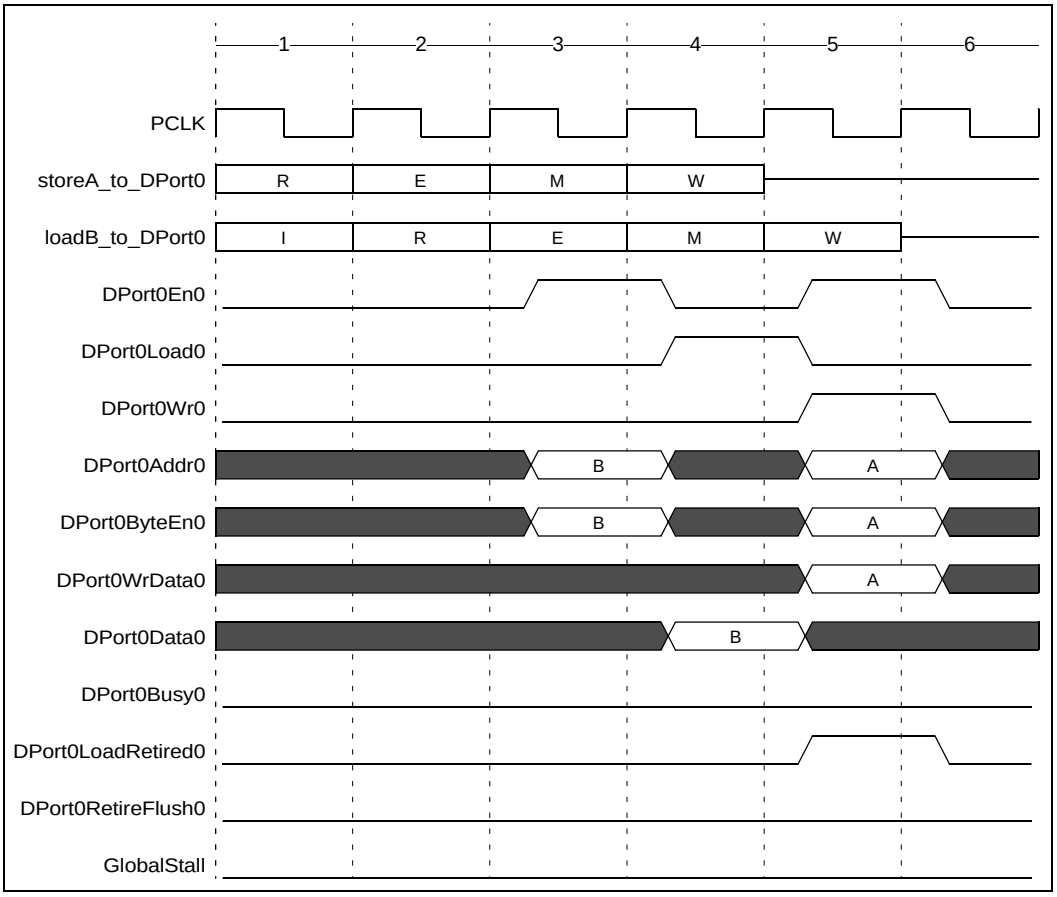


Figure 20–116. Store and Speculative Load (5-Stage Pipeline)

Similarly, Figure 20–117 shows a write transaction initiated at the same cycle that RetireFlush is signalled. In all cases, the XLMI device must correctly handle both the load-related processing and the write.

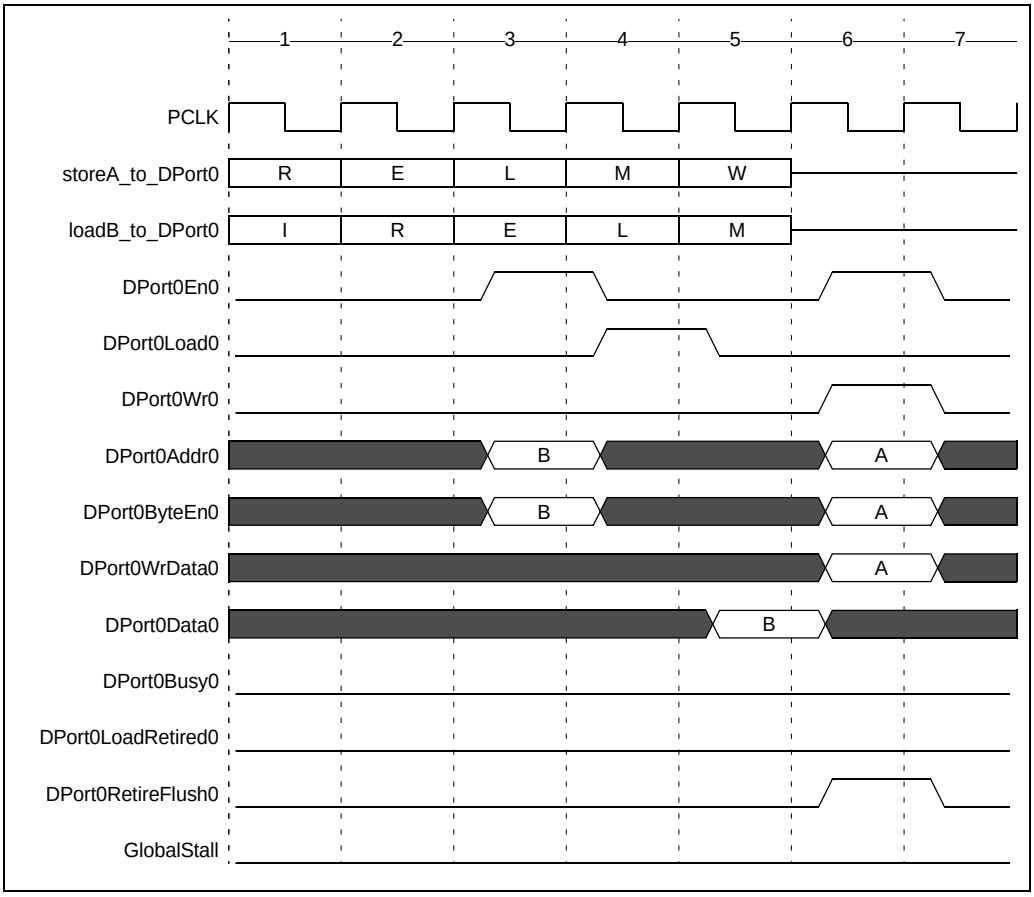


Figure 20–117. Store and Speculative Load (7-Stage Pipeline)

20.4.5 XLMI Port Compared to Queues Interfaces

External FIFO queues can also be connected to the XLMI port. To choose between the XLMI port and queue interfaces (to connect to external FIFOs) consider the following points (in addition to other system constraints):

- FIFO queues connected to the XLMI port are accessed via memory-mapped load and store instructions. External queues attached to the processor through the QIF32 IPQ and OPQ queue interfaces must employ the READ_IPQ and WRITE_OPQ instructions, because the queue interfaces do not exist within the processor's address

space. The designer should consider whether memory-mapped or queue push and pop operations are the more appropriate usage model. For some system designs, the interface choice won't matter.

- If a FIFO queue is attached to the XLMI port, it must share that port with any other XLMI-attached devices, which may result in bandwidth issues. Queue interfaces (QIF32) are not shared, and therefore an external queue attached to a queue interface receives the full bandwidth of that interface.
- If the processor tries to read from an input queue when it's empty, an XLMI-attached FIFO queue will stall the I/O operation. The stalled I/O read will stall the processor's pipeline as well. This stalled state is not interruptible. Queues can stall the processor's pipeline but interrupts can still occur when a queue is stalled.
- If the processor executes a store to the XLMI port immediately followed by a load from an XLMI-attached FIFO queue, the store is buffered and the load occurs first. If this load causes an I/O stall because the addressed FIFO queue is empty, a resource deadlock will freeze the processor's pipeline. The XLMI load cannot complete because the FIFO queue is empty and the store to the FIFO queue cannot take place because the pipeline is stalled.
- All XLMI-attached devices, including FIFO queues, must handle speculative reads over the XLMI port. To do this, XLMI-attached devices must observe and use the processor control signals that indicate whether a read has been committed or flushed. The processor automatically handles speculative reads for (QIF32) input-queues.

21. Xtensa ROM Interface Ports

21.1 Instruction ROM and Data ROM Interface Ports

The Xtensa ROM Interface ports resemble the RAM interface ports. They are fast, memory-mapped, local-memory interfaces with guaranteed availability. However, ROM ports only read data from the attached memory: the Xtensa processor cannot write data to a ROM port.

The ROM Interface ports guarantee that instructions or data are provided to the processor in the shortest time possible. A synchronous instruction or data ROM with 1-cycle latency (for a 5-stage pipeline) or 2-cycle latency (for a 7-stage pipeline) is connected to the port and is mapped to a contiguous address range whose start address must be an integer multiple of its size. The instructions or data are accessed when the fetch or load/store address calculated by the processor core falls within the port's configured address range. The processor can have one instruction ROM and one data ROM.

As with the cache and RAM interface ports, the ROM port's address and control signals are presented during one cycle, and the attached ROM returns the requested data either one cycle later (5-stage pipeline) or two cycles later (7-stage pipeline).

Processor accesses to both the instruction- and data-ROM interface ports can be blocked by the use of the `Busy` signals: `IRom0Busy` and `DRom0Busy`*m* (see Section 21.2 “ROM Port Signal Descriptions” on page 400 for signal-naming conventions). These signals indicate that during the previous clock cycle, the corresponding instruction or data ROM was unavailable. Such a lack of availability can occur because the ROM is being accessed by the other load/store unit (for Xtensa processor configurations with two load/store units), by an external entity such as a DMA controller, or by another processor core. This mechanism allows several processors to share one ROM. More details on `Busy` signal behavior and how the signal can be used appear in Section 18.3 “Instruction RAM Data-Transfer Transaction Behavior” on page 296, Section 18.9 “Data RAM Transaction Behavior” on page 314, and Section 23.3 “Memory Combinations, Initialization, and Sharing” on page 439.

An Inbound-PIF transaction is not applicable to instruction or data ROM. As with data RAM, the data ROM can be configured as 1, 2, or 4 banks. Banking is implemented in the same way as data RAM (see Section 18.6 “Banking for Local Memories” on page 311), except that data cannot be written to a ROM port.

When multiple banks are configured, the Xtensa processor generates a C-Box to support its functionality accordingly.

Note: Instruction ROM cannot have multiple banks.

If two load/store units are configured, load sharing is supported in hardware (this requires C-Box configuration). The implementation is the same as load/store sharing for data RAM (see Section 18.8 “Load/Store Sharing” on page 312), except that the ROM port does not have write capability.

Memories and other devices with read side-effects (such as a FIFO) must not be connected to ROM interface ports because of the speculative nature of Xtensa processor read operations. Xtensa processor loads are *speculative*: a read operation to ROM does not necessarily mean that the processor will consume the read data and retire the load instruction. For performance reasons, many loads will attempt to read data from the ROM port even though the actual load target address is assigned to another memory interface. A variety of internal events and exceptions can cause pipeline flushes that subsequently cause the processor to replay loads targeted at addresses assigned to the ROM ports, because the data obtained from the first load has been flushed. Consequently, all devices attached to an Xtensa processor’s ROM ports must accommodate the speculative nature of all processor read operations.

21.2 ROM Port Signal Descriptions

Table 10–20 and Table 10–16 list and describe the data- and instruction-ROM interface port signals. The signal-naming convention includes the memory name, memory number denoted by *n*, and (for data memories) the load/store unit number *m* or bank ID. For example, `DRom0Addr1` is the address to DataROM0 driven by load/store unit 1. (Note, *n* is always= 0 for instruction and data ROM ports. The load/store unit number *m* is nonexistent when the C-Box option is selected for single-bank data ROMs.) Zero or one instruction-ROM ports and zero or one data-ROM ports can be configured. Some of the ROM port signals are not present unless other configuration options are selected in addition to the instruction ROM or data ROM options. These signals are shown in the tables in the Notes column.

Note: See Section 35.8 on page 633 for information about ROM interface AC timing.

21.3 ROM Data-Transfer Transaction Behavior

Instruction and data ROM behavior resembles instruction and data RAM behavior except that the processor cannot initiate a store operation to local ROM. An attempt to store to ROM causes an exception as shown in Table 21–93. See Section 18.3 “Instruction RAM Data-Transfer Transaction Behavior” on page 296 for more details.

Table 21–93. RAM/ROM/XLMI Access Restrictions

Memory or Port	Instruction Fetch	L32R L32I L32I.N	Other Loads[1]	S32I S32I.N	S32C1I	Other Stores[2]
InstROM	ok	ok[3, 6]	LSE[5]	LSE[5]	LSE[5]	LSE[5]
InstRAM	ok	ok[3, 6]	LSE[5]	ok[3, 6]	LSE[5]	LSE[5]
DataROM	IFE[4]	ok	ok	LSE[5]	LSE[5]	LSE[5]
DataRAM	IFE[4]	ok	ok	ok	ok	ok
XLMI	IFE[4]	ok	ok	ok	LSE[5]	ok

1. All other load opcodes, including 32-bit TIE loads

2 All other store opcodes, including 32-bit TIE stores

3 Reduced performance.If executed as part of a FLIX instruction, causes a Load Store Error exception if the operation is not issued from slot 0

4. Instruction Fetch Error exception

5. Load Store Error exception

6. In a multi-slot FLIX instruction bundle, load/store operations directed at instruction ROM and RAM must occupy slot 0 of the instruction bundle, and in the case of two load/store units, it must not be paired with a second, simultaneous, load or store. Otherwise, a load/store exception will occur.

   Also see the Note in Section 18.9.4 about LoadStoreErrorCause exceptoins.

21.3.1 Data ROM and the C-Box

The Xtensa C-Box option allows a processor configuration with the multiple load/store units to have just one interface to each data RAM or data ROM. For more information about the C-Box option, see Section 18.5 “Different Connection Options for Local Memories” on page 308.

22. Xtensa Processor System Signals and Interfaces

This chapter provides notes on various Xtensa signals and interfaces:

- Clock
- Reset
- Wait mode
- Runstall
- Processor ID signals
- Static vector select
- Interrupts
- ErrorCorrected signal

Note: See Chapter 35 for information about AC timing.

22.1 Xtensa Processor Configuration Options for System Interfaces

This section covers the following configuration options:

- PIF Bridge
- Relocatable Vectors:
  The relocatable vectors option allows the position of the reset, exception, and interrupt vectors to be dynamically relocated. By default, the reset and memory error handler vectors can be in one of two statically configured locations, as selected by the `StatVectorSel` pin.
- Processor ID:
  Optional 16-bit input to give the processor a unique ID for multi-processor systems.
- Interrupts and High-Priority Interrupts:
  The Xtensa processor can have as many as 32 interrupt signals from external sources. Each interrupt line can be configured as level- or edge-triggered.
- Non-maskable Interrupt:
  The Xtensa processor can have one Non-Maskable Interrupt (NMI) signal.

22.2 Clocks

Clock related signals are listed in Table 22–94.

Table 22–94. Clock Signals

Name)	I/O	Description	Notes
CLK	Input	Clock input to the processor	
BCLK	Input	Bus clock when core and bus clocks are asynchronous.	Requires Asynchronous AMBA bus option. Additionally requires either AHB-Lite or AXI bridge.
Strobe	Input	Indicates phase of the bus clock to support synchronous clock ratios between the core and bus	Requires AHB-Lite or AXI bridge, and not asynchronous AMBA bus option.
JTCK	Input	Standard 1149.1 JTAG TAP clock.	Requires OCD or PSO
PBCLK	Input	APB clock.	Requires OCD or PSO
ATCLK	Input	Unused.	Required ATB option of TRAX

CLK, BCLK, JTCK and PBCLK are asynchronous input clocks to the Xttop module and are distributed throughout the design. Some clocks, namely CLK, BCLK and PBCLK may be gated before they are used by leaf flops. Except for the clock gating logic that uses latches, the Xtensa design uses all edge-triggered flip-flops. Given that fact, the Xtensa IP does not impose any duty cycle requirements on any of its clocks. The implementation scripts in the hardware package specify a 50% duty cycle, but you may choose any duty cycle that meets timing in your sign-off flow.

The processor core and the system bus interface clock have one of three different relationships:

1. The processor and bus clocks are synchronous and the same frequency. Both are clocked by the CLK input.

2. The processor and bus clocks are synchronous, but the processor clock frequency is an integer multiple faster than the bus frequency. Multiples up to a 4-to-1 clock ratio are allowed. The bus interfacing logic is clocked with the same CLK input as the core, but an additional `Strobe` input is used to signal rising edges of the slower bus clock. Figure 22–118 shows the relationship of the clocks and Strobe input for each of the four supported clock ratios.

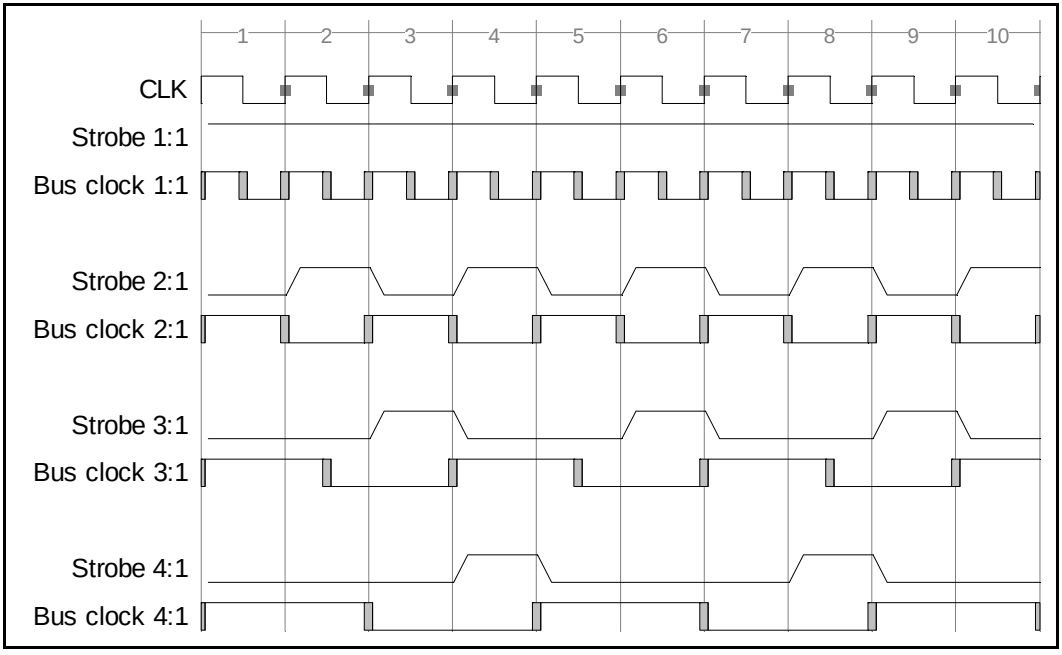


Figure 22–118. Clock Strobe

3. The processor and bus clocks are asynchronous. An asynchronous FIFO synchronizes PIF requests and responses. The bus interfacing logic uses the bus clock, BCLK. The bus clock is allowed to be slower or faster than the processor clock, however the faster of the two clocks must not have a frequency greater than 16 times the frequency of the slower clock. In addition, the core and bus clocks must be truly asynchronous to one another for correct functionality.

JTCK is the input clock for the Access Port and Debug Module. The frequency of CLK must be at least three times the frequency of JTCK. CLK and JTCK must be asynchronous to one another.

PBCLK is the input clock to the Access Port and Debug Module when APB is configured. There is no frequency requirement between CLK and PBCLK. CLK and PBCLK must be asynchronous to one another.

ATCLK is a clock that is part of the ATB. The ATB is an optional interface that is implemented when TRAX is configured with ATB support. ATB must be clocked synchronously to Xtensa i.e. ATCLK be the same as CLK, therefore ATCLK is unused within Xtensa.

Note: In PSO configurations, CLK, BCLK and PBCLK must always be toggling because the PCM relies on counting clocks to wakeup individual domains. Wakeup will be delayed for as long as there are no edges of the clocks of the given domain.

22.3 Reset

This section describes Xtensa processor reset fundamentals, including the different sources of reset, full vs. partial reset, and synchronous vs. asynchronous reset structure.

22.3.1 Sources of Reset

There are multiple sources of reset to the Xtensa processor:

- HW Pin Reset: These are pins at the Xttop and Xtmem level, the assertion of which cause all or portions of Xtensa logic to be reset.
- SW-Controlled Reset: These are bits in AccessPort registers, the setting of which cause portions of Xtensa logic to be reset.
- PSO Reset: In a manner not directly observable by the outside system, various portions of Xtensa logic are reset when the domain containing the logic is woken out of PSO by the PCM. These resets are present only in configurations with PSO.

Note: Figure 8–24 on page 91 shows a high-level block diagram including Xtmem and Access Port.

HW Pin Reset

Table 22–95 lists the HW pins that are related to reset.

Table 22–95. Reset Signals

Name	I/O	Description	Notes
BReset	Input to Xttop and Xtmem	Reset input to the processor. Active high.	Resets the Xtensa core.
DReset	Input to Xttop and Xtmem	Reset input to the Debug Module. Active high.	Resets debug functionality such as OCD, TRAX and Performance Counters. Requires OCD.
PReset	Output of Xttop	BReset synchronized to CLK. Active high.	Resets local memory data staging registers. Not used in 5-stage pipelines. Used in 7-stage pipeline configurations to reset memory data staging registers.
JTRST	Input to Xttop and Xtmem	JTAG interface reset. Active low.	Resets JTCK flops primarily in AccessPort, such as TAP FSM. Requires OCD or PSO.
PRESETn	Input to Xttop and Xtmem	APB interface reset. Active low.	Requires OCD or PSO. [Not to be confused with PReset. Historical use of PReset and requirements of APB spec has lead to unfortunate similarity of names.
ATRESETn	Input to Xttop and Xtmem	ATB interface reset. Active low.	Requires ATB option of TRAX.
PcmReset	Input to Xttop and Xtmem	Reset input to Power Control Module. Active high.	Requires PSO.

ATRESETn is a reset that is part of the ATB. Xtensa resets the ATB using DReset and not ATRESETn. ATRESETn is used only for data gating (i.e. for power reduction) of the interface.

PcmReset is a reset to the Power Control Module alone, and does not go to Xttop. It is only present in configurations with PSO. PcmReset is used for Full Reset only as described in the first paragraph below. Assertion of PcmReset in any other manner will lead to undefined behavior.

SW-Controlled Reset

The PWRCTL register in the Access Port has two bits related to reset.

- CoreReset: Setting this bit asserts reset to the Xtensa core and core-related logic at the Xttop level. The bit must be explicitly cleared in order for reset to be deasserted.
- DebugReset: Setting this bit asserts reset to debug-related logic in Xttop, such as many portions of the Debug Module. The bit must be explicitly cleared in order for reset to be deasserted.

More details regarding these bits is available in the *Tensilica On-Chip Debugging Guide*.

PSO Reset

In a manner not directly observable by the outside system, various portions of the Xttop logic are reset when the domain containing the logic is woken out of PSO by the PCM. The number of these internal resets is roughly equivalent to the sum of the number of clock domains per power domain. The PCM ensures that assertion requirements are met such that the domain is properly woken up - again in a manner transparent to the user.

These resets are present only in configurations with PSO, and without state retention.

22.3.2 Full vs. Partial Reset

A full reset is necessary at the start of use of an Xtensa-based system. All functions within Xtmem/Xttop, including the processor core, memories, debug, etc., are reset during a full reset. Subsequent sections of this chapter describe the procedure required to accomplish a full reset.

Once full reset has been completed, it is possible to selectively reset some portions of Xtensa without affecting the reset state of others. Partial reset of the following functions is supported:

- The processor core and the system bus interface
  - Using the BReset signal or
  - Using a control bit in the AccessPort PWRCTL register
- Debug functions (i.e., OCD, TRAX and Performance Counters)
  - Using the DReset signal or
  - Using a control bit in the Access Port PWRCTL register
- The JTAG interface
  - Using JTRST
- The APB interface
  - Using PRESETn

All clocks must be toggling normally during the entire period of partial reset.

When a full reset is conducted, all outstanding transactions to Xtensa are aborted. Similarly, partial reset also results in aborted transactions. Some examples are:

- outstanding inbound PIF request to core that is partially reset
- issuance of an OCD interrupt to Debug which is partially reset

The full reset or partial reset requirements described above guarantees only that all the components of Xtmem/Xttop are reset correctly. There may be additional requirements on the resets if certain functional features are to be used. Sections relevant to the functional feature will describe these in more detail.

22.3.3 Synchronous vs. Asynchronous Reset

You can configure the Xtensa processor to use either synchronous or asynchronous reset. A system designer generally decides upon a synchronous or asynchronous reset methodology based on the target library and implementation flow. For example, some user-design methodologies may favor an asynchronous reset methodology because:

- The target library may not include synchronous reset registers.
- Immediately upon the assertion of reset, Xtensa outputs will be in a known state. (In synchronous reset configs, outputs will be undefined till the reset propagates through the flops of the reset tree.)
- Gate level simulations of asynchronous reset configurations may have fewer X-propagation issues.

In contrast, some user-design methodologies may favor a synchronous reset methodology because:

- Synchronous reset implementation flows can be simpler. Reset may be treated as a data input, rather than a special global signal.

Offering both options allows the processor to conform to the user's design requirements. Each configuration option has different characteristics, but the same sequence requirements in order to properly accomplish reset. The sequence is described in sections below.

Note: Irrespective of the reset style chosen, all flops clocked by JTCK (e.g., the TAP controller flops), are asynchronously reset as per the IEEE 1149.1 specification.

The next few sections describe requirements that the system must meet in order to properly reset the Xtensa processor. Inputs and outputs refer to pins at the Xtmem module boundary. Note that "defined" means that a signal will not have X or Z value.

22.3.4 Full Reset of an Asynchronous-Reset Xtensa

Following are the steps in resetting and bringing correctly to operation an asynchronous-reset configuration:

1. Assert (as present in the configuration) PcmReset, JTRST, PRESETn, BReset and DReset.

2. Hold these for at least the time that the assertion edge needs to propagate through the reset combinational fan-out, to the leaf-state elements and outputs of the Xtensa processor.

   This time can be ascertained from post-place-and-route timing reports for each reset signal.

3. Deassert the resets.
   - Full reset is now complete, and all Xtensa outputs are guaranteed to be defined.
   - There is no requirement for the resets to be deasserted at the same time.
4. Ensure all inputs to the Xtensa processor have defined values, before the first clock edge is given to Xtensa.
   - The clocks in question here are (as present in the configuration) CLK, BCLK, PBCLK and JTCK. This case of reset without clocks is shown in Figure 22–119.
   - If clocks had been running through reset assertion, inputs must have defined values prior to the deassertion of reset. This case of reset with clocks is shown in Figure 22–120.
5. All Xtensa outputs will continue to be defined from this point onward, as long as inputs also remain defined.

PCLK, the first signal in the waveforms of Figure 22–119 and Figure 22–120, is simply a representation of the slowest of CLK, BCLK and PBCLK.

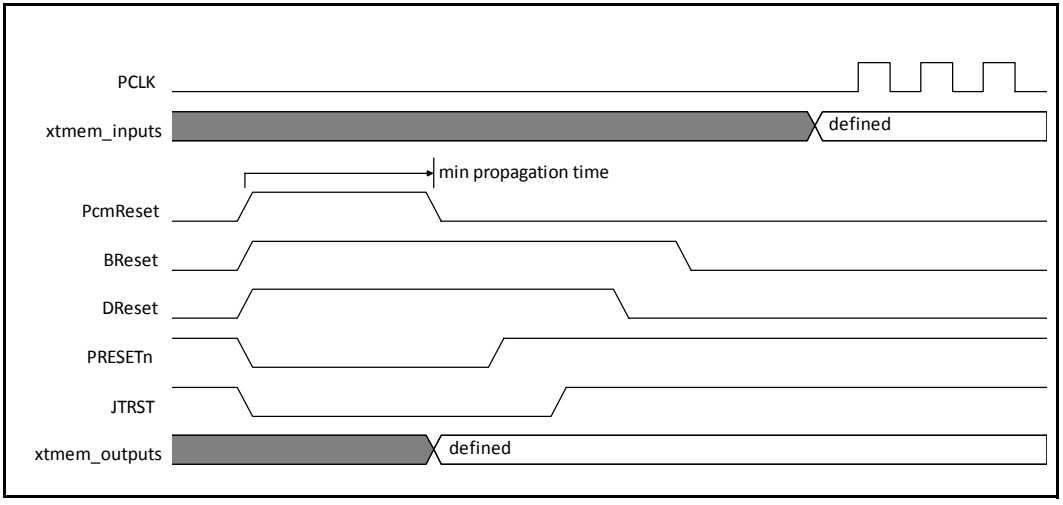


Figure 22–119. Full Reset: Asynchronous Reset Without Clocks

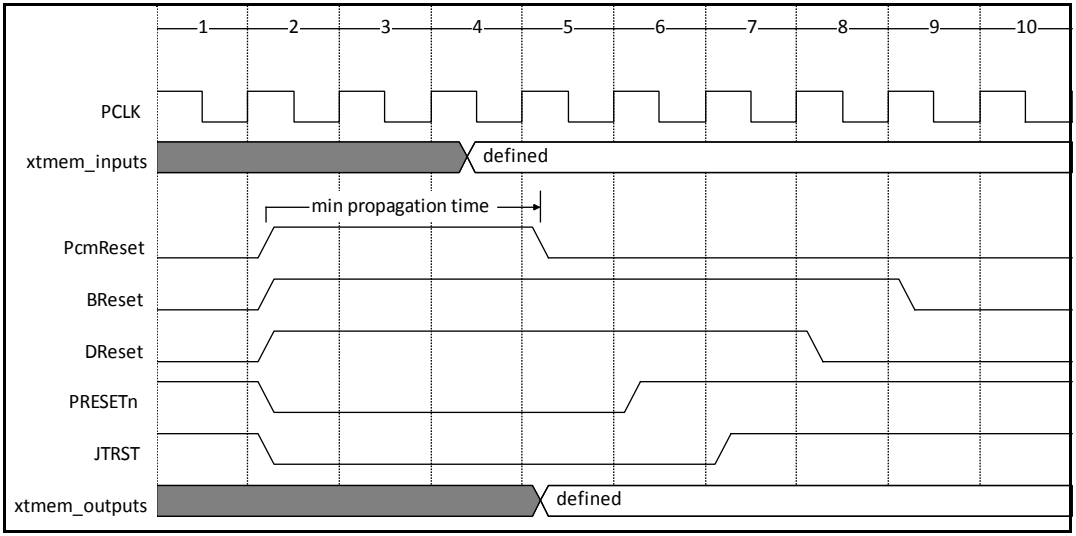


Figure 22–120. Full Reset: Asynchronous Reset with Clocks

Note that Figure 22–120 shows a minimum propagation time that is roughly three clock cycles. This is simply an example. The period of a clock and the minimum propagation time required for a reset signal are independent variables.

22.3.5 Full Reset of a Synchronous-Reset Xtensa

Following are the steps in resetting and bringing correctly to operation a synchronous-reset configuration:

1. Assert (as present in the configuration) PcmReset, JTRST, PRESETn, BReset and DReset.
2. Hold these asserted while providing at least ten rising edges of each of CLK, BCLK and PBCLK.
3. Deassert the resets.
   - Even though this is a synchronous-reset configuration, JTRST is an asynchronous reset as required by the IEEE JTAG specification. It is not required for JTCK clock edges to be provided for the proper reset of JTCK flops. However, JTRST must still be held long enough for the assertion edge to propagate through combinational fan-out to leaf state elements.
   - Full reset is now complete, and all outputs of Xtensa are guaranteed to be defined.
   - There is no requirement for the resets to be deasserted at the same time.

4. Ensure all inputs to Xtensa have defined values before any subsequent clock edges are given to Xtensa.

   The clocks in question here are (as present in the configuration) CLK, BCLK, PBCLK and JTCK.

5. All Xtensa outputs will continue to be defined from this point onward, as long as inputs also remain defined.

Figure 22–121 shows the waveform for this reset case.

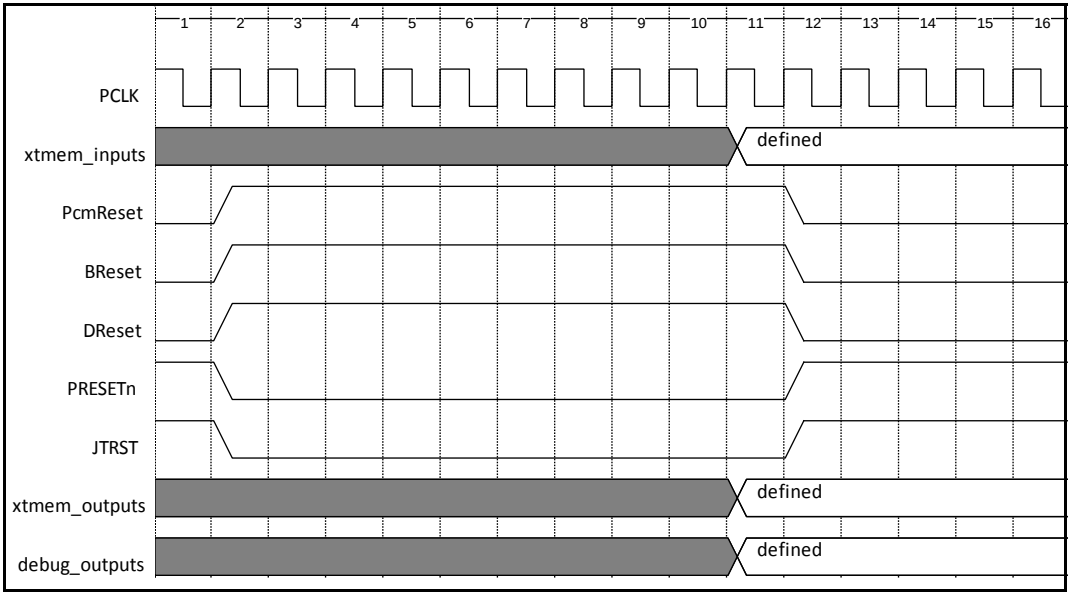


Figure 22–121. Full Reset: Synchronous Reset

22.3.6 Partial Reset

The partial reset sequence is similar regardless of function. In other words, whether using, for example DReset, to reset the debug functionality or PRESETn to reset APB logic or BReset to reset the core, the basic sequence required to perform the reset is the same. Following are steps for DReset only.

The difference between full reset and partial reset is that inputs are expected to remain defined through the partial reset sequence. Taking the synchronous-reset Xtensa case, following are the steps to do a partial reset of debug logic.

1. Ensure that all inputs to Xtensa have defined values.

   This is a given in that full reset must have happened to properly bring up Xtensa, and subsequently inputs must remain defined for proper operation.

2. Assert DReset.

3. Hold asserted while providing at least ten rising edges each of CLK, BCLK and PB-CLK.
4. Deassert DReset.
5. Partial reset is now complete, and debug registers and outputs will have their reset values.

The following figure shows the waveform for this reset sequence. The first signal in this figure, PCLK, is a representation of the slowest of CLK, BCLK and PBCLK.

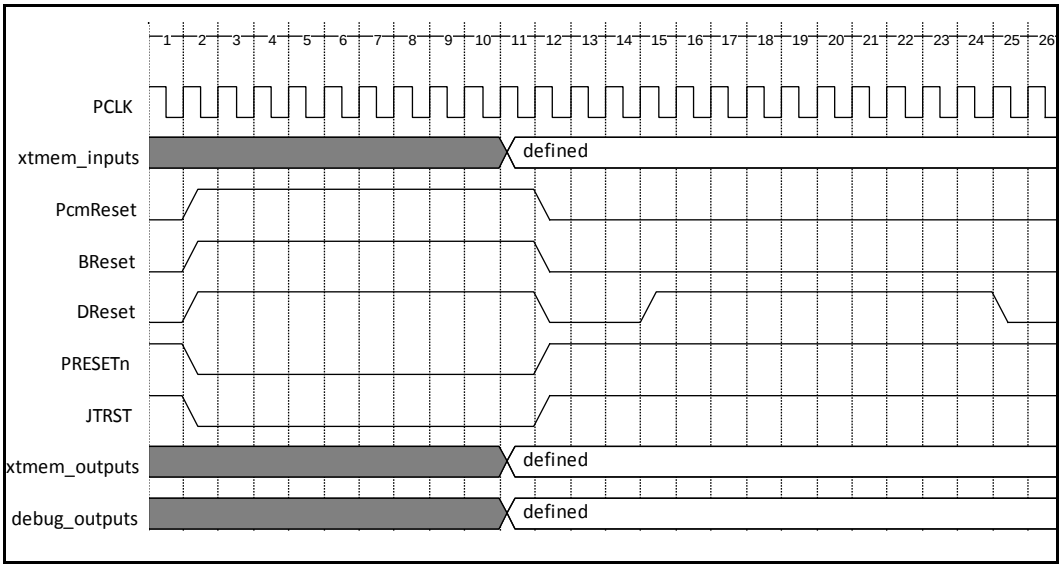


Figure 22–122. Partial Reset of Debug Logic in Synchronous Reset Configuration

22.3.7 Inputs with Specific Reset Requirements

Certain input signals have specific requirements with regard to reset as described in this section. The requirements are the same for both asynchronous-reset and synchronous-reset configurations.

AXIPARITYSEL

This is a signal that is sampled by the AXI bus bridge logic at reset. It must be held for 10 cycles of the slower of CLK and BCLK after BReset deassertion.

PRID, StatVectorSel, and AltResetVec

These are inputs that are sampled by the core during its bring-up. They must be held for 10 CLK cycles after BReset deassertion.

OCDHaltOnReset

To put Xtensa into OCD mode via the assertion of OCDHaltOnReset, BReset deassertion must occur 10 CLK cycles after DReset deassertion.

The rules above are always followed when doing a full reset; but when doing a partial reset, signals not associated with the function being reset are not subject to the requirements. For example, AXIPARITYSEL is not Debug-related, therefore it does not have to be held as described above during Debug partial reset.

22.3.8 Reset Signal Synchronization to Multiple Clocks

Figure 22–123 shows why in the sequence above, BReset must be held asserted for ten rising edges of both CLK and BCLK. When the asynchronous AMBA bus option is enabled, the reset input (BReset) is synchronized into the core clock domain (CLK) and the bus clock domain (BCLK) as shown in Figure 22–123. For the bridge-side logic, any reset pulse length requirements must be met relative to the bus clock, rather than the core clock.

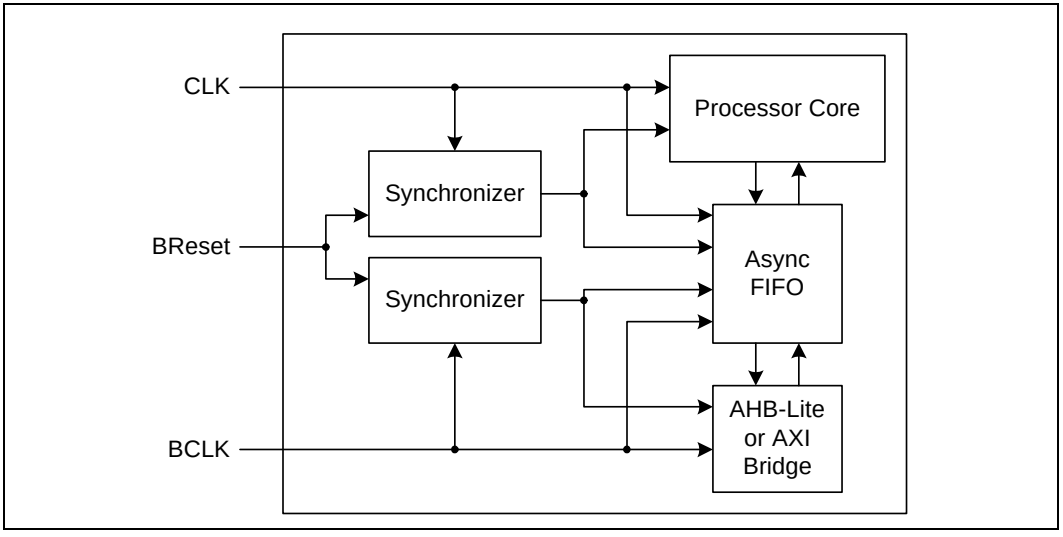


Figure 22–123. Reset Synchronizers in Configs with Asynchronous AMBA Bus Option

Table 22–96 shows the full matrix of resets. The BReset CLK synchronizer of Figure 22–123 corresponds to the row 1 column 1 entry of the table. Similarly, the BReset BCLK synchronizer corresponds to the row 2, column 1 entry of the table.

Table 22–96. Sources of Reset

Flops	HW Reset	SW Controlled Reset	PSO Reset
All CLK flops in Processor Domain	BReset pin	CoreReset bit of PWRCTL	Proc domain wakeup
All BCLK flops in Processor Domain	BReset pin	CoreReset bit of PWRCTL	Proc domain wakeup
CLK flops in the Debug Domain that talk to the Processor	BReset pin	CoreReset bit of PWRCTL	Debug domain wakeup OR Proc domain wakeup
CLK flops in the Debug Domain that talk to APB	DReset and PRESETn pins ORed	DebugReset bit of PWRCTL	Debug domain wakeup
CLK flops in the Debug Domain that talk to JTAG	DReset and JTRST pins ORed	DebugReset bit of PWRCTL or TAP FSM in its Reset state	Debug domain wakeup
CLK flops in the Debug Domain that do not talk to Core, APB or JTAG	DReset pin	DebugReset bit of PWRCTL	Debug domain wakeup
All PBCLK flops in the Debug Domain	PRESETn pin	N/A	Debug domain wakeup
All JTCK flops in the Debug Domain	JTRST pin	TAP FSM in its Reset state	N/A

Each row of Table 22–96 has at least one independent synchronizer. The synchronizer is generally shared across the columns.

22.3.9 RunStall and Local Memory Busy at Reset

Note that all busy signals from the instruction RAM, data RAM, and XLMI ports, as well as the run/stall signal (see Section 22.6 on page 417) can be active during and immediately following the assertion of reset, and have no restrictions relative to reset other than that they be stable and known. This feature can be useful for initializing RAMs at reset time. To initialize memories with the inbound PIF option, the following sequence is defined:

1. The RunStall input must be asserted two cycles before the deassertion of BReset. This stops the processor from attempting to fetch instructions immediately after BReset is deasserted.

2. Inbound PIF requests can be issued ten cycles after the deassertion of BReset. The processor drives POReqRdy to a low value during reset, and will assert POReqRdy to a high value when it is ready to begin accepting requests.
3. Deassert RunStall after all inbound PIF requests have completed. Configurations with write responses can wait for the final write response.

22.4 Static Vector Select

The Relocatable Vectors option adds the `StatVectorSel` pin to the processor's set of I/O pins. The `StatVectorSel` pin specifies the addresses to be used for the processor's Reset and (if configured) Memory Error Handler vectors. Using the Relocatable Vectors option without the External Reset Vector sub-option, both the Reset and Memory Error Handler vectors will exist in one of two static locations selected when the processor is configured. Driving a 0 on `StatVectorSel` selects the default vector locations, and driving a 1 on `StatVectorSel` selects the alternate locations.

Note that when the Memory Error option is also configured, there is an alignment restriction on the Reset Handler vector location. Specifically, the location address must have a sufficient number of lower address bits zeroed, equaling the smallest power-of-2 which is large enough to contain the full extent of either the Memory Error Handler vector or the Reset Handler vector. When the Memory Error option is not configured, such an alignment restriction does not apply, but other Xtensa technology requirements may restrict the alignment of the Reset Handler vector location.

`StatVectorSel` must be held stable for 10 CLK cycles following the deassertion of `BReset`. Changing the input prior to this point produces unpredictable results. `StatVectorSel` is sampled at reset time only, so changing the input subsequent to this point has no effect on vector locations.

22.4.1 External Reset Vector Sub-Option

When the Relocatable Vectors option is selected, this External Reset Vector sub-option becomes available. This sub-option adds an input bus of pins to the Xttop module, called `AltResetVec[31:0]`. The contents of this input bus are sampled exactly once per `BReset` in the same mechanism and timing as `StatVectorSel`, (as above, Section 22.4) so changing this input bus afterwards has no effect on vector locations. Additionally, the same alignment requirements noted above also apply to this input bus.

Whether or not this sub-option is selected, driving a 0 on `StatVectorSel` selects the default static vector location determined during processor configuration. However, with this sub-option selected, driving a 1 on `StatVectorSel` selects the sampled contents

of the AltResetVec[31:0] input bus. Note that, if this sub-option is not selected, driving a 1 on StatVectorSel selects (as above, Section 22.4) the alternate reset vector assigned during processor configuration.

22.5 Wait Mode

The wait mode option adds the PWaitMode port and WAITI instruction to the processor core. Wait mode is always enabled as part of the interrupt option. When the WAITI instruction completes, it sets the interrupt level in the processor's PS.INTLEVEL register to the value encoded in the instruction, waits for all of the processor memory interfaces to become idle, and causes the processor to assert the PWaitMode signal. Processor operations suspend until a non-masked interrupt occurs.

If the global clock-gating option is enabled, then the processor generator will add gating to the majority of the clocks in the processor. Clock gating saves power while PWaitMode is asserted. See Chapter 34, "Xtensa Processor Core Clock Distribution" on page 599 for more details on clock distribution within the processor. If global clock gating is not enabled, then processor operations will be suspended while PWaitMode is asserted, but no power-saving mechanism will be active inside the processor. No other non-reset action besides a non-masked interrupt will bring the core out of wait mode once PWaitMode has been asserted.

The processor remains in low power mode, asserting PWaitMode output, until an interrupt (which was enabled prior to executing WAITI) is received. Any enabled interrupt will "wake up" the processor. Instruction fetch resumes a small number of cycles after the interrupt is received. Instruction fetch (e.g. IRam/ICache enable assertion) will not occur while PWaitMode is asserted, although IRam/ICache may be enabled for I-fetch in the same cycle PWaitMode is de-asserted. In general, PWaitMode is not appropriate for controlling the power of other SoC components, particularly IRam. SoC designers should also note that IRam enables may be asserted during PWaitMode due to Inbound PIF requests.

Inbound-PIF requests are not considered processor operations and continue while the processor is in wait mode. The inbound-PIF-request logic's clock network is gated by a separate set of conditions from most of the other processor logic, to allow inbound-PIF requests to continue functioning while the processor is in wait mode.

22.6 Run/Stall

The RunStall signal allows an external agent to stall the processor and shut off much of the clock tree to save operating power.

22.6.1 Global Run/Stall Behavior

The RunStall input allows external logic to stall the processor's internal pipeline. Any in-progress PIF activity such as instruction- or data-cache line fills will complete after the assertion of the RunStall input line.

Inbound-PIF requests to the local instruction and data RAMs will still function even though the pipeline has stalled.

The RunStall input can be used

- as a mechanism to initialize local instruction and data RAMs using inbound-PIF requests after a reset. If the RunStall input is asserted during reset and remains asserted after the reset is removed, the processor will be in a stalled state. However inbound-PIF requests to local instruction and data RAMs can still occur. The Instruction and data RAMs can be initialized using inbound-PIF requests and after these RAMs have been initialized, the RunStall can be released, allowing the processor to start code execution. This mechanism allows the processor's reset vector to point to an address in local instruction RAM, and the reset code can be written to the instruction RAM by an external agent using inbound-PIF write requests before the processor starts executing code.
- to freeze the processor's internal pipeline. Assertion of RunStall reduces the processor's active power dissipation without using or requiring interrupts and the WAITI option. However, the power savings resulting from use of the RunStall signal will not be as great as for those using WAITI, because the RunStall logic allows some portions of the processor to be active even though the pipeline is stalled. (See Chapter 32, "Low-Power SOC Design Using Xtensa Processors" on page 561 for more information on low-power design using the processor.)

RunStall signal timing appears in Figure 22–124. The processor synchronizes the RunStall input internally so that the pipeline stalls one cycle after RunStall is asserted. The GlobalStall signal shown in the diagram is an internal processor signal that gates much of the processor's clock tree. The resulting gated clock appears as G1SCLK in the figure. The processor's pipeline restarts two clock cycles after RunStall is negated.

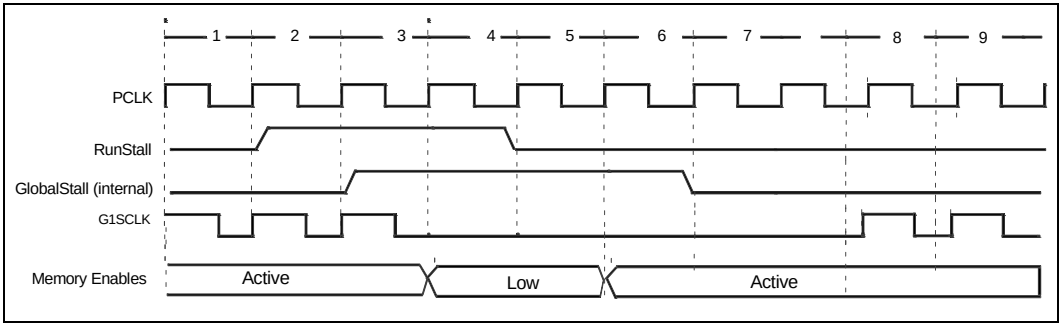


Figure 22–124. RunStall Behavior and Timing

The extra stall cycle following the negation of the RunStall input allows reactivation of the local-memory enables. To save power, memory enables are deactivated during RunStall-initiated pipeline stalls (from 2 cycles after RunStall is asserted to 1 cycle after RunStall is deasserted).

There are exceptions for certain cases, including:

- A cache-line fill is outstanding or in progress when RunStall is asserted, The cache memories will be enabled as required to complete the cache line fill.
- Inbound-PIF requests are being made to local instruction or data RAMs. These memories will be enabled as required to respond to the inbound-PIF requests.
- Stores are being retired from the store buffers. Local memories will be enabled as required.
- A load that received a Busy signal will continue to try and access the memory until the memory is no longer Busy.

22.7 Processor ID

Some applications employ multiple Xtensa processors that execute from the same instruction memory so there is often a need to distinguish one processor from another in these situations. The processor ID (PRID) allows the system logic to provide each Xtensa processor with a unique identity by adding a 16-bit input bus, PRID, as an input to the processor core. When the processor comes out of reset, this input is latched into the low 16-bits of special register 235, which can be read by an RSR.PRID instruction.

The PRID port is not a dynamic input. The processor only latches the value of the PRID port at reset time. The PRID input must be stable for 10 cycles following the deassertion of BReset. Changing the PRID input before this point produces unpredictable results. Changing the PRID input after this point has no effect on the PRID register.

Note: Future Xtensa processor implementations may have wider or narrower PRID input ports.

22.8 Error Corrected Signal

When ECC is configured for any of the Xtensa processor's data-RAM, data-cache, instruction-RAM, or instruction-cache interfaces, the ErrorCorrected output signal is present to indicate when the processor hardware has corrected a correctable memory. This output pin is asserted when the first instruction after the processor hardware corrects a correctable memory error commits and it and remains asserted until either software or another memory error event has cleared the Recorded Correctable Error (RCE) bit of the Memory Error Status Register(MESR).

The ErrorCorrected output signal is useful in certain applications where real time response is crucial and the logging and correcting of memory errors must not increase the response time of high-priority code. By connecting this output pin to one of the Xtensa processor's level-sensitive interrupt pins, an assembly-level interrupt routine can be written and used to service or log memory expeditiously (that is, with low interrupt latency). Use of the ErrorCorrected output pin only applies to correctable errors when the DataExc bit of the MESR is not set. Uncorrectable errors will still cause an exception. In such cases, real-time response may not be possible.

22.9 Interrupts

The external system uses BInterrupt[iwidth-1:0] to signal interrupts to the Xtensa processor, where iwidth is the configured number of external interrupt signals. Each interrupt input can be configured to be level- or edge-triggered. Assertion of an interrupt input causes code execution to jump to the appropriate interrupt handler routine at a specified vector location. One of the interrupt bits can be configured by the designer to be a non-maskable interrupt (NMI).

22.9.1 Level-Triggered Interrupt Support

Level-triggered interrupts are set and cleared by the external source of the interrupt. The time required for the processor to take an interrupt varies widely and depends on many factors. Therefore, external logic that generates level-triggered interrupts should assert and hold the interrupt line active until the interrupt is serviced by the software.

Note: The Xtensa processor's interrupt-vector model allows many interrupts to share the same interrupt vector and service handler.

22.9.2 Edge-Triggered Interrupt Support

A single-cycle pulse synchronous to PCLK on an interrupt line from an external source signals an edge-triggered interrupt. The processor latches the interrupt on the rising edge of PCLK. The edge-triggered interrupt condition is cleared by writing to a special register.

22.9.3 NMI Interrupt Support

One interrupt input can be designated as the non-maskable interrupt. This interrupt is sampled like an edge-triggered interrupt as described above but it cannot be masked by software and has a separate interrupt vector (as do high-priority interrupts).

Because the Xtensa processor does not mask the NMI, the time period between successive assertions of NMI should not be so short as to interrupt the processor while it is saving state from a previous NMI. Such an occurrence will prevent the processor from being able to return to the user code that was executing when the first NMI was triggered. When calculating the amount of time the processor needs to save its state prior to starting an NMI service routine, the designer should include memory-system latencies, which will affect the time required to store processor-state values. For processes or events that will generate frequent interrupts, the designer should use a high-level interrupt instead of the NMI and should assure that the interrupt is properly unmasked as soon as the absolute required state has been saved.

Use extreme caution when allowing the system to assert the processor's NMI input while the processor is executing its reset code. Non-maskable interrupts can interrupt reset code just as it can any other code, which means that the reset code might be interrupted before the NMI vector or interrupt-handling code is initialized in memory. This caveat is especially important for versions of the Xtensa processor with the relocatable vector option because the interrupt can occur even before the code has initialized the processor's vector base register. Using an uninitialized vector or vectoring to uninitialized memory instead of an interrupt-handling routine will produce undefined results, which is very undesirable. In general, systems should not let an NMI occur until system initialization occurs but there may be circumstances, such as debugging situations, where it's advantageous to let NMIs occur during the execution of reset code. Xtensa processors allow NMIs to occur during reset code for such situations but the wary system designer will use this feature with extreme caution.

22.9.4 Example Core to External Interrupt Mapping

Table 22–97 illustrates an example of an Xtensa processor's core-to-external interrupt signal mapping. (**Note:** Numerous other interrupt mappings are possible. Table 22–97 merely shows one possible configuration as an illustration.) `BInterrupt[0]` is associated with the lowest order bit of the interrupt register that is configured as an external in-

terrupt (an edge-triggered interrupt, level-triggered interrupt, or a non-maskable edge-triggered interrupt). For the example shown below, bit 1 is the lowest-order bit in the interrupt register to be associated with an external interrupt because bit 0 has been configured as a software interrupt. `BInterrupt` input lines with increasingly higher numbers are associated with increasingly significant bits in the interrupt register, if those bits are configured for external interrupts.

In this example, bit 2 of the interrupt register is assigned to one of the Xtensa processor's internal timers, so the `BInterrupt[1]` interrupt input is associated with the next-most significant bit assigned to an external interrupt, which in this example is bit 3 of the interrupt register. Similarly, the `BInterrupt[2]` interrupt input is associated with bit 4 of the interrupt register.

Table 22–97. Example Core to PIF External Interrupt Mapping

Interrupt/Enable Bit	Interrupt Type	BInterrupt Bit
0	Software	
1	External Edge	`BInterrupt[0]`
2	Timer	
3	External Level	`BInterrupt[1]`
4	External Edge	`BInterrupt[2]`
5	Software	

22.10 Interrupt Handling

Xtensa processors can have as many as 32 level- or edge-triggered interrupt sources. The states of these interrupt sources are latched in a special 32-bit register, named the `INTERRUPT` register. Each `INTERRUPT` register bit can be independently configured to indicate external, software, or Internal interrupt classes.

External interrupts are connected to the `BInterrupt` input pins and can be configured to be one of three types:

- Level-sensitive
- Edge-triggered
- NMI (edge-triggered non-maskable interrupt)

Note: Only one NMI interrupt signal is allowed.

Software interrupt bits are readable and writable by software, which uses the `INTERRUPT` register address to read, set, and clear a software interrupt.

Internal interrupt bits are set by processor hardware (that is, timer logic) and can only be read by software, which can clear the interrupt indirectly by writing to the associated source of the interrupt (that is, a timer control register).

Note: See Section 13.3 for information about Xtensa processor bus errors and associated exception behavior.

External logic that generates interrupts should be synchronized with PCLK (the internal processor clock, see Section 34.3) to ensure that the interrupt request lasts at least one clock cycle and to avoid metastability problems by ensuring that the interrupt request meets the setup-time requirements detailed in Chapter 36.

Level-sensitive interrupt inputs are latched directly in the INTERRUPT register. The processor will continue taking the associated interrupt exception as long as a BInterrupt level-sensitive input is asserted. Like the internal interrupts, external level-sensitive interrupts must be cleared at the source.

Edge-triggered interrupt inputs must be asserted for at least one processor clock cycle to ensure detection. After the corresponding interrupt bit is set, it will remain set even when the external signal is negated. Software can read these edge-triggered interrupt indicator bits by reading the INTERRUPT register and can clear these bits by writing to the INTCLEAR register (special register 227).

Note: Software cannot set external interrupt bits in the INTERRUPT register.

Interrupt-enable bits are defined by another special register, named the INTENABLE Register, which is readable and writable. The INTERRUPT register bits are bitwise AND'ed with the corresponding bits of the INTENABLE register. The resulting bits are then all OR'ed together to form an internal interrupt signal that can cause an interrupt exception.

To reiterate, edge-triggered, internal, and software interrupt sources can set the associated interrupt bits in the INTERRUPT register, but only software can clear these bits using the INTCLEAR register. Level-sensitive interrupt sources directly set and clear the associated interrupt bits in the INTERRUPT register.

Designers can also configure any of the interrupt signals to be a NMI (non-maskable interrupt). A bit in the INTERRUPT register designated as the NMI bit will not be masked by the corresponding INTENABLE register bit.

23. Local-Memory Usage and Options

Xtensa processor local memory consists of caches, local data memory, and local instruction memory. The processor also has a fast local memory port (XLMI) that can be used to connect the processor to additional local memories or to fast, memory-mapped peripheral blocks.

23.1 Cache Organization

Caches are fast and small memories that are used to buffer the processor from slower and larger memories located external to the SOC. The Xtensa processor core's caches store the data and instructions that a program is currently using, while the rest of the data resides in slower main memory external to the processor core. In general, the Xtensa processor can access the separate instruction and data caches simultaneously to maximize cache-memory bandwidth.

The Xtensa processor accesses cache information when there is a *hit* in the cache look-up sequence. Xtensa processor caches are physically tagged. A hit occurs when there is a match between one of the cache's physical-address tags and the physical target address. A *miss* occurs when the physical address tags of the cache do not match the physical target address. A miss means that the target instruction or data is not present in the cache and that the cache line must be retrieved from the slower external memory.

A cache line is one entry in the cache (per way for set-associative caches). For example, as illustrated in Figure 23–125, each way of a 4-way set-associative cache has its own cache line, which is indexed and used if hit and refilled if a load or store causes a cache miss.

When there is a cache hit, the Xtensa processor fetches the target instruction or data from the appropriate cache and the operation continues. A read cache miss initiates a cache-line refill sequence involving a pipeline replay and the generation of a request for new data from external memory before the operation can continue.

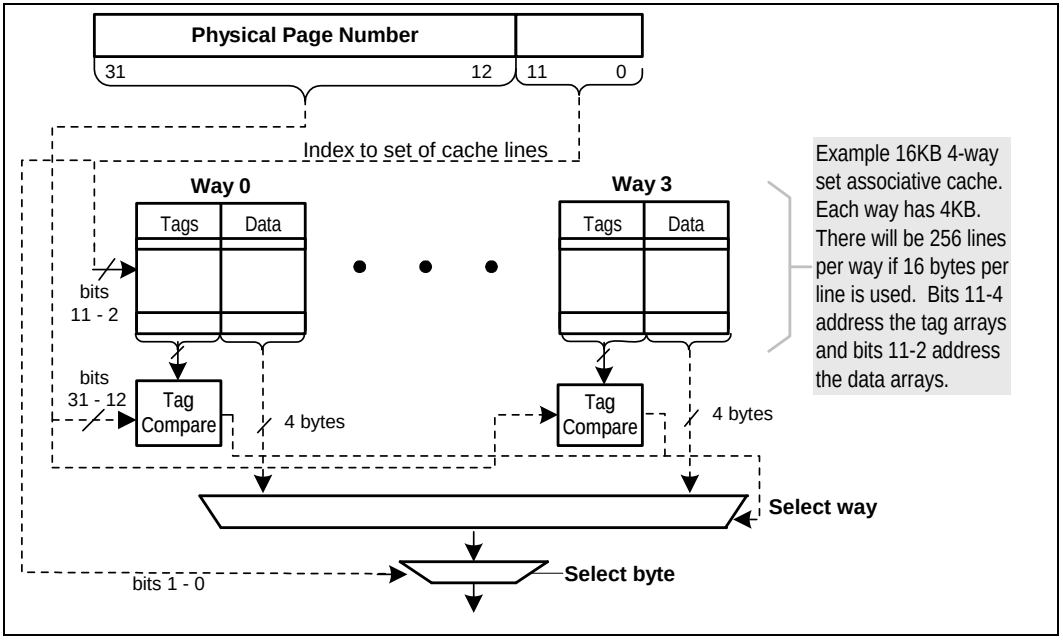


Figure 23–125. Example 4-way Set-Associative Cache Organization

23.1.1 Organization of Instruction Cache

The instruction cache has the following configurable characteristics:

- cache size
- cache-line size
- set-associativity
- virtually indexed
- checked with a physical tag
- optional lockable on a per-line basis
- optional parity or ECC memory-error protection
- ability to dynamically enable and disable cache ways for power savings

As shown in Figure 23–126 and Figure 23–127, the instruction-cache tag-array width depends on the configured data-array size, the choice of set associativity (line replacement algorithm bit appears depending on configuration), the cache write-policy choice (a dirty bit is present when write-back is configured), and the presence or absence of line locking.

Note: Although the instruction-cache data array actually holds instructions, the array is referred to as a data array to distinguish it from the instruction cache's tag-field array.

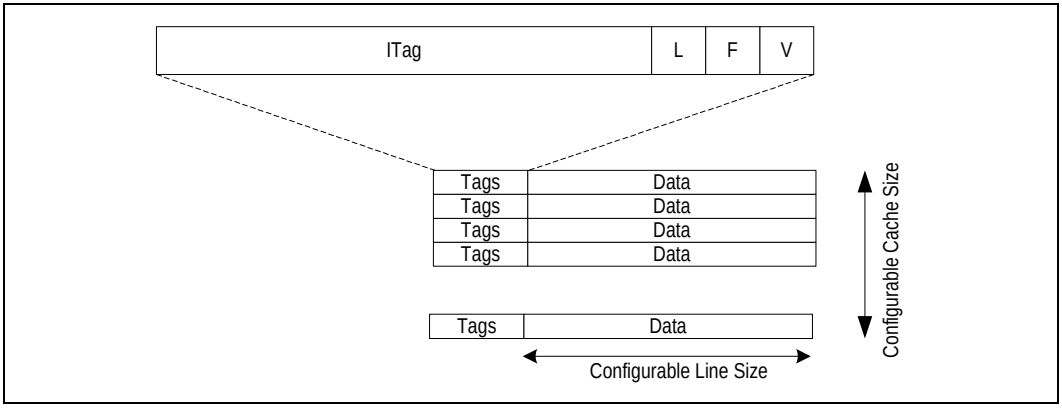


Figure 23–126. Tag versus Data Array: Example Instruction Tag Array Format

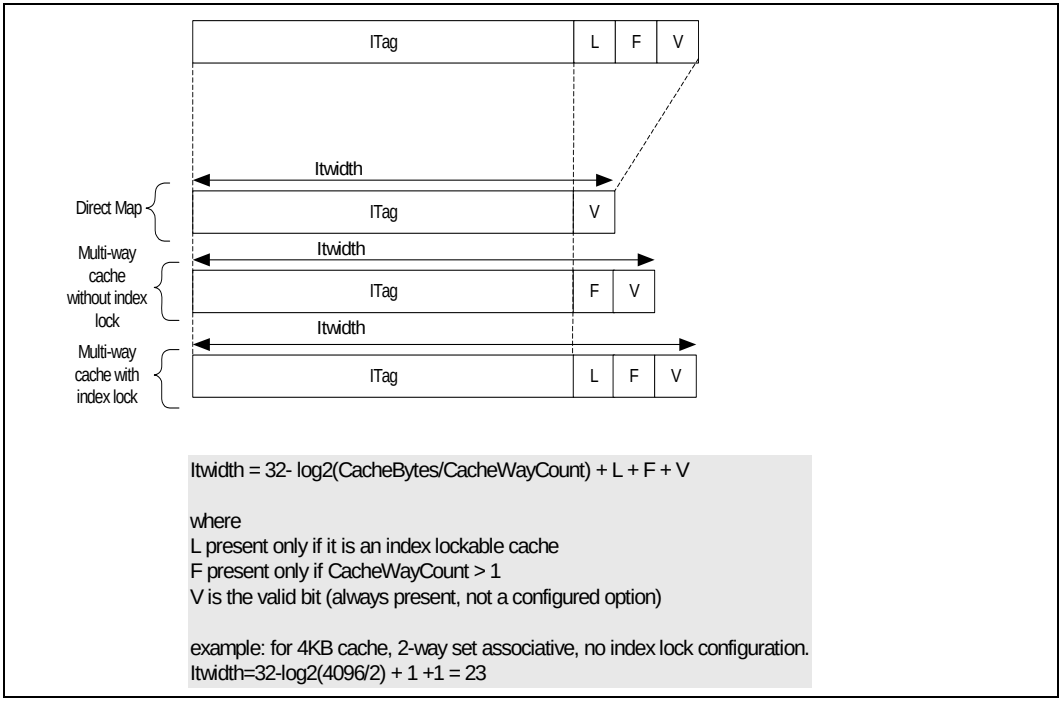


Figure 23–127. Instruction Cache Tag Physical Memory Structure

Figure 23–127 illustrates the physical memory structure of the instruction cache tag. Table 23–98 describes the cache-tag fields.

Table 23–98. Instruction Cache Tag Field Descriptions

Field	Description
V	Valid bit. If the instruction tag is valid, then V is set to 1.
F	Least-recently-filled algorithm bit (for multi-way caches only). This bit is stored with the tag for each cache line/way. The line-replacement algorithm bit is used for cache-miss refill way selection.
L	Lock bit per line (locking configurations only). If the L bit is 1 and the V bit is 1, then the associated line is excluded from the cache-miss refill way selection.
ITag	Instruction tag. Physical address tag of the instruction cache used to match a physical address or the TLB's physical page number.

Note: ITag, V, F and L are un-initialized after reset, or power-on reset. Software must reset before use.

23.1.2 Organization of Data Cache

The data cache has the following configurable characteristics:

- cache size
- cache-line size
- set associativity
- write-through or write-back cache write policy (write-back write policy must be configured when more than one load/store unit is configured)
- virtual address indexing
- checked with a physical tag
- optional locking on a per-line basis
- optional parity or ECC memory-error protection
- number of cache banks (when more than one load/store unit is configured)
- ability to dynamically enable and disable cache ways for power savings

The tag array width depends on the configured data array size, the choice of set associativity (the line-replacement-algorithm bit appears or disappears depending on configuration), the cache write-policy choice (a dirty bit is present when write-back is configured), and the presence or absence of line locking.

Figure 23–128 describes the structure of the physical data-cache memory. The data-cache tag fields are described in Table 23–99.

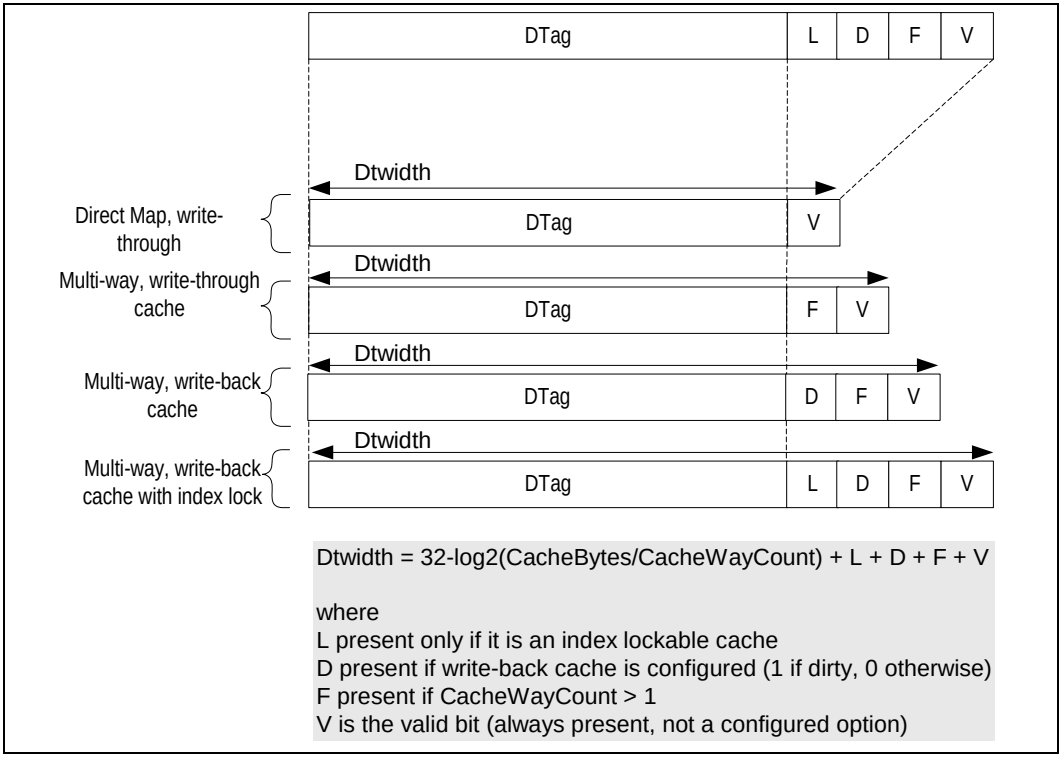


Figure 23–128. Data Cache Tag Physical Memory Structure

Table 23–99. Data Cache Tag Field Descriptions

Field	Description
V	Valid bit. If the data cache line is valid, then V is set to 1.
F	Least-recently-filled algorithm bit (for multi-way caches only). This bit is stored along with the tag for each cache line or way. The line-replacement-algorithm bit is used for cache-miss refill way selection.
D	Dirty bit (for a write-back configuration only). A line selected for replacement (victim) is cast out if it is dirty, regardless of its cache attribute. If the data-cache line has been modified, then D is set to 1.
L	Lock bit per line (locking configurations only). If the L bit is 1 and the V bit is 1, then line is excluded from cache-miss refill way selection.
DTag	Data tag. Physical address tag of the data cache used to match the physical address or the TLB's physical page number.

Note: DTag, V, F, D, and L are un-initialized after reset, or power-on reset. Software must reset before use.

23.1.3 Data-Cache Write Policy

The same data can be kept in the main memory and in cache. Data held in both the cache and main memory is kept consistent using either a write-through or a write-back policy.

For a write-through cache configuration, a write operation to the data cache is always accompanied by a write of the same data to main memory. Memory is always coherent with the cache using this scheme. However, there may be a performance impact, because write-through caches increase processor-to-main-memory bus traffic.

Note: If a line is not already in cache, no allocation occurs, and the write is simply sent to memory.

For a write-back cache configuration, the processor core may defer writing values to main memory after writing them to the data cache. Each cache line contains a dirty bit indicating whether or not main memory needs to be updated with the values in the associated cache line. For a data-cache miss, the processor must evict a data-cache line to make room in the cache for the newly required line. If the line to be evicted from the cache is dirty, the processor core must first write the old line to main memory before bringing a new line into the cache. The processor's write buffer is used for data-cache-line write-back operations.

The Xtensa processor's writeback cache can either use a write-allocate or a no-write-allocate policy. When the write-allocate policy is selected, if a store causes a cache miss, the processor must first load the appropriate cache line into the cache from main memory before writing the new value into the cache. However when the no-write-allocate policy is selected, a store miss will not cause a cache miss or a cache fill. Instead the store will be written directly to the PIF.

Note: When configuring a data cache with two load/store units, the data cache must be a write-back cache configuration. The write-back cache may be operated as write-through by programming it with write-through cache attributes.

Compared to write through caches, write-back caches generate less main-memory bus traffic when the same location is written multiple times over a short time period (the code sequence is said to exhibit temporal locality). However write-back caches with the write-allocate policy enabled may increase the bus traffic relative to write-through caches for streaming writes of large data sets. If streaming writes are required, a no-write-allocate policy may be desirable to avoid unnecessary cache refills.

The characteristics of the Xtensa processor's data-cache write policy include:

- Write-back attributes are selected with the TLB.
- Attributes are configurable on a per-page basis.

- In write-back mode, the castout sequence completes in one cycle if the PIF width equals the cache-line size. The castout sequence requires more than one cycle if the PIF width is smaller than the cache-line size.
- The line-replacement algorithm is *Least Recently Filled*. Cache lines that are marked as ‘invalid’ are given the highest replacement priority.
- Caches can be configured as direct-mapped or 2-, 3-, or 4-way set-associative.
- For a write-through cache configuration, the write to memory may not occur at the same time as the write to cache because of write-buffer limitations.
- When a cache line is selected for replacement (victim), either in the course of satisfying a cache miss or during the execution of a write-back or write-back-invalidate instruction, the processor writes the victim line back to memory (castout) if the cache line is valid and has been modified (dirty bit is set). The dirty bit can be set if any portion of the address space has ever had the write-back attribute.
- For a write-through entry, whenever a store hits in a cache line, the data written to the cache is also written to main memory through the write buffer. The store will not set or clear the dirty bit for a write-through cache line to allow a different virtual address that maps to the same physical address using a write-back policy to still set the dirty bit. The result is that the dirty bit will be set if any byte of the cache line has been written without updating the corresponding byte of main memory.
- Allocation
  - Write-through: Never allocate on a store miss
  - Write-back: Always allocate on a store miss
  - Write-back allocate: Always allocate on a store miss
  - Write-back no allocate: Never allocate on a store miss

23.1.4 Line Locking

Cache-line locking allows the processor to lock critical code or data segments into the instruction or data caches on a per-line basis. This feature allows the system programmer to maximize the efficiency of the system design. The cache-locking function is a selectable option on the Xtensa Processor Generator, and supports both data-cache locking and instruction-cache locking mechanisms.

The Xtensa processor’s direct-mapped cache does not support cache line locks.

Instruction-Cache Line Locking

The `IPFL` instruction locks an instruction-cache line. The `IHU` and `IIU` instructions unlock instruction-cache lines.

The IPFL instruction locks a line for the specified address into the instruction cache. If the specified address is not in the instruction cache, then the line is automatically transferred from memory to instruction cache. The IPFL instruction initiates the instruction-cache-line fill if the IPFL fill data is not already present in the cache. The processor pipeline stalls until the line fill completes. At least two unlocked instruction-cache ways must be available for proper cache-line locking. Otherwise, the IPFL instruction causes an exception.

Once locked, the line behaves like a line in local instruction RAM, and the locked lines are not replaced when an instruction-cache miss occurs. Lines may be unlocked using the IIU or IHU instructions. Invalidate instructions IHI and III do not invalidate locked instruction lines.

Data-Cache Line Locking

The counterpart instructions that are involved in data-cache locking include DPFL, DHU, and DIU.

The DPFL instruction locks a line for the specified address into the data cache. If the specified address is not in the data cache, then the line is automatically transferred from memory to the data cache. The cache line-locking option requires more than one way of associativity depending on the configuration. The number of ways of cache must be greater than or equal to the number of load/store units + 1. Because the Xtensa processor does not pre-fetch data, the DPFL instruction is treated as a normal load. At least two unlocked data-cache ways must be available for proper cache line locking. Otherwise, the DPFL instruction will cause an exception. Figure 23–129 illustrates the different possibilities for a 2-way, set-associative cache. A DPFL instruction will succeed in locking a cache line only if both ways of the set it is trying to use are initially unlocked. The DPFL instruction causes an exception if either of the ways it is trying to use is already locked.

Once locked, the cache line will behave like a line in local data RAM, and the locked lines will not be replaced when a data cache miss occurs. Lines may be unlocked using the DIU or DHU instructions. Invalidate instructions DHI and DII do not invalidate locked data-cache lines.

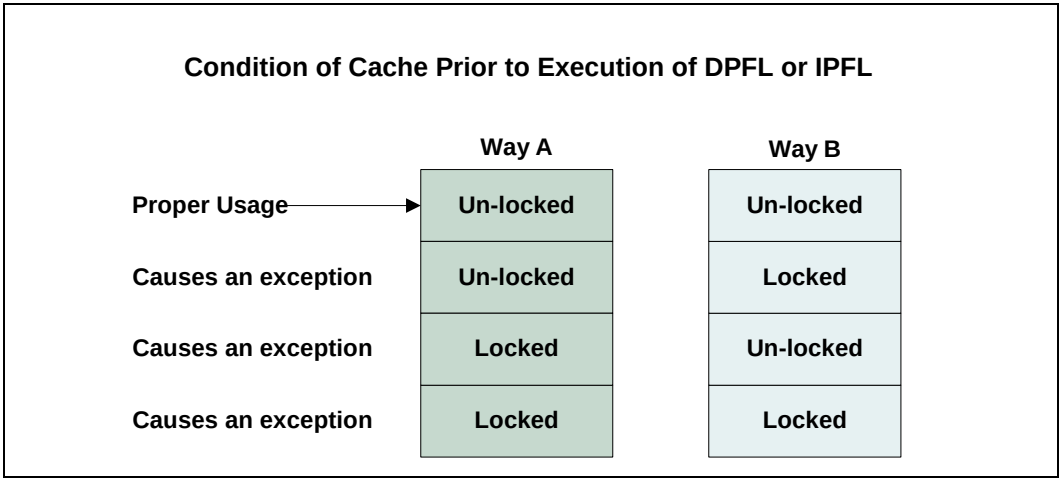


Figure 23–129. Cache Locking Example (for 2-Way Set-Associative Cache)

Silent Line Unlocking

Silent cache-line unlocking will occur in the instruction or data caches if all ways of a cache somehow become locked. This safeguard is necessary because the processor core requires unlocked ways for proper operation of the line-replacement algorithm. For example, it is possible for software to lock all the data-cache ways through the inappropriate use of SDCT instructions. Should this happen, the Xtensa processor will silently unlock the first way of the data and instruction caches and no exception will occur. Table lists the cache-line-locking instructions and their functions.

Cache Line Lock Instructions

Instruction Cache	Data Cache	Function
IPFL	DPFL	Cache line is locked per address.
IHU	DHU	Un-lock cache line, if it hits.
IIU	DIU	Un-lock cache line at specified indexed way.

Note: SICT and SDCT can lock and un-lock cache line. However, these are reserved manufacturing test instructions and should be used with caution when used for line locking.

23.1.5 Dynamic Cache Way Usage Control

The Dynamic Cache Way Usage Control feature allows the programmer to dynamically disable and enable cache ways for power saving using new and modified instructions.

Configuration Restrictions

- If this option is to be configured, it needs to be configured for all of the implemented caches on a given processor. You can configure either all of the caches or none of them.
- We can only specify using ways 0 through X (where X <= N-1) of an N way cache (currently N=1,2,3,4)No random combinations of ways in use are supported

New State and Instruction Functionality

The purpose of this feature is to allow the processor to do a graceful shutdown of any number of present Instruction-Cache and Data-Cache Ways to facilitate a power saving mode, and the subsequently re-enable of those that were shut down.

There are new state bits in MEMCTL (SR97) to indicate how many Cache ways are in use. There is new functionality added to the xsr and wsr.MEMCTL instruction that will cause invalidation of indicated cache ways if the number of used ways specified by the instruction is greater than the ways currently in use. There is a new instruction, DIWBUI.P, for use on the DCache that will write back (if dirty), unlock, and invalidate a cache line specified by a tag way and index.

Instruction-Caches and Data-Caches are treated differently on shut down but similarly on re-enable.

- Instruction-Cache Ways can be disabled simply by marking the disabled ways as unused in the MEMCTL register, see Table 23–100. This will disable Cache Enables, ignore hits, and disable allocation for future re-fills in those ways
- Data-Cache Ways can be disabled in a 3 step process:
  - **Disable allocation** for future re-fills in the ways being disabled
  - Using a new unlock/invalidate/dirtyWB instruction (DIWBUI.P), **start a process of dirty line cast outs and line invalidates** for the ways that are being disabled. This is a software loop.
  - **Mark the disabled ways as unused**, shut off Cache Enables, and ignore all tag data from them.
- Instruction-Cache and Data-Cache Ways can be re-enabled as follows:
  - Using a single **xsr or wsr.MEMCTL instruction** with its new functionality, **issue an interruptible string of invalidate operations** to all of the indexes of the ways (in parallel) being re-enabled (I or D or both)
  - When that process is complete, **mark the previously disabled ways as usable** and allow allocation and hits in them.

New Fields for MEMCTL State Register SR(97)

Table 23–100. Dynamic Cache Way Usage fields in MEMCTL State Register

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
MEMCTL[12:8]	1	5	Data-Cache Ways in Use	R/W	97
MEMCTL[17:13]			Data-Cache Ways Allocatable	R/W	97
MEMCTL[22:18]			Instruction-Cache Ways in Use	R/W	97
MEMCTL[23]			Invalidate Enable (Normally set to '1')	R/W	97

1) Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions.

The three 5-bit fields in Table 23–100 are in place to facilitate disabling individual ways of a cache. The numbers represent how many Ways are in use or allocatable. Zero means none. If the value is greater than or equal to the number of ways in the processor, then all ways are enabled.

For example, if "Data-Cache Ways in Use" is set to three in a four way Data-Cache configuration, then only ways 0, 1, and 2 will be in use. Way 3 will not normally be signalled in any way by the processor. It will only access that way(s) when an xsr or wsr.MEMCTL instruction is used to re-enable that way(s) of the cache.

If "Data-Cache Ways Allocatable" is set to a number less than the maximum number of ways in the configuration, then new misses will not be allocated to that way(s) but data currently in a non-allocatable way will still be in use. Typically this state is entered so that the new DIWBUI.P instruction can be used to cast out dirty data to clean the Way before it is disabled.

xsr or wsr.MEMCTL New Functionality

The wsr.MEMCTL (and xsr.MEMCTL) instruction has new functionality to handle the new fields listed in Table 23–100. (For the remainder of this section it will simply be referred to as the w/xsr.MEMCTL instruction meaning both wsr or xsr.).

This instruction and the MEMCTL State Register are the main elements of this configuration option. These elements are controlling two independent caches and thus the execution of one w/xsr.MEMCTL instruction can cause two independent operations to occur in parallel. The instruction will finish and proceed down the pipe only when the longer of the two operations has finished.

If the w/xsr.MEMCTL instruction is decreasing the number of ways of a cache, it does the following:

- Wait for the cache(s) it is targeting to finish any outstanding operation from Load/Store, Instruction Fetch, or Prefetch.
- Write the new (lesser) value to either MEMCTL[12:8] (Data-Cache Ways in Use) or MEMCTL[22:18] (Instruction-Cache Ways in Use)
- Do a Replay of the next instruction so that the new Cache state is in effect immediately.

When decreasing the number of ways of a write-back configured Data-Cache, the user should be careful to make use of the "Data-Cache Allocatable" field of MEMCTL and the new DIWBUI.P instruction, which can be used to unlock and cast-out dirty lines thereby preserving data integrity for the Cache. See Section “DIWBUI.P New Instruction” for more details.

If the w/xsr.MEMCTL instruction is increasing the number of ways of a cache, it does the following:

- Wait for the cache(s) it is targeting to finish any outstanding operation from Load/Store, Instruction Fetch, or Prefetch.
- Do a series of tag invalidate writes to every tag in the re-enabled ways. This is an interruptible instruction see Section “List of Special Cases for this Feature” for more information
- Write the new (greater) value to either MEMCTL[12:8] (Data-Cache Ways in Use) or MEMCTL[22:18] (Instruction-Cache Ways in Use)
- Do a Replay of the next instruction so that the new Cache state is in effect immediately

Note that when increasing the number of ways, in order to have the invalidates occur, the w/xsr.MEMCTL has to be setting MEMCTL[23] "Invalidate Enable". This bit is put here for testability. If it is not set then the instruction will simply enable the new ways of the cache(s) without any invalidates. The user would be responsible for ensuring that none of these newly enabled ways were used before proper initialization took place.

When increasing the number of Data-Cache ways, the user should increase the "Data-Cache Ways Allocatable" field of MEMCTL to the same value in either the same w/xsr.MEMCTL instruction or a subsequent one.

If the w/xsr.MEMCTL instruction is changing the number of ways that are allocatable in the Data-Cache it does the following:

- Wait for the Data-Cache to finish any outstanding operation from Load/Store or Prefetch
- Write the new value to MEMCTL[17:13] (Data-Cache Ways Allocatable) or MEMCTL

- Do a Replay of the next instruction so that the new Cache state is in effect immediately

See Section “DIWBUI.P New Instruction” for more details. The Data-Cache Ways Allocatable field is used in conjunction with that new instruction to clean out dirty lines in a Data-Cache while not allowing any new tags to be allocated in the ways being brought down.

Finally, if the MEMCTL fields for either Cache are not being altered by this w/xsr.MEMCTL, then it is treated as a NOP with respect to these fields of the MEMCTL register. Note that this instruction could still be used for other fields in this register. For example, the above listed stalls and replays will not be forced on a w/xsr.MEMCTL instruction that is only altering the L1 Memory Controls MEMCTL[0].

It is perfectly fine to have one w/xsr.MEMCTL instruction increasing one Cache while decreasing the other. One w/xsr.MEMCTL instruction can alter one cache or both. Any combination will give the desired results.

DIWBUI.P New Instruction

The new instruction, called DIWBUI.P is used during DCache Way Reduction. DIWBUI.P works like a combination of a DIU and a DII in as much as it unlocks and invalidates a tag at a particular index but only does it for the specified way and if the line is dirty, initiates a write back.

A replay upon completion of the DIWBUI.P instruction will happen if the line it is invalidating is dirty and thus requires a write back.

The DIWBUI.P instruction will use the same way/index format in AR as the DIWBI instruction.

Upon completion of the DIWBUI.P, AR will be incremented. The increment will extend and wrap through the Way field.

List of Special Cases for this Feature

- **The Tag*En and Tag*Wr signals** for disabled ways are inhibited for all instructions except when a w/xsr.MEMCTL is invalidating them.
- **Interrupts:** The w/xsr.MEMCTL instruction can potentially take quite a few cycles and hence was made interruptible. Microstate will be saved if a w/xsr.MEMCTL instruction is interrupted while increasing, and thus invalidating the tags of some cache ways. After the interrupt(s) is(are) completed, if an identical w/xsr.MEMCTL is executed, it will simply pick up where the interrupted one left off. If it is a w/xsr.MEMCTL instruction where the any of the new MEMCTL fields are different than the interrupted one, it will start over with the new instruction.

- **RunStall:** When "RunStall" is asserted in the middle of a w/xsr.MEMCTL invalidate sequence, it will not stall the remainder of the invalidates. It will stall the instruction from advancing in the pipeline once it has finished its invalidate sequence.
- **Maximum Number of Ways:** If the value of the number of Ways in the MEMCTL SR is greater than the maximum number of ways, it is considered to be equal to the number of ways for the purposes of the wsr.MEMCTL instruction.
- **Allocatable but not In Use:** If the user sets a way as "Allocatable" without being enabled as "In Use", then that way is considered not "Allocatable" and not "In Use" by the hardware.
- **PIF and AXI Attributes:** PIF and AXI attributes are not impacted when all cache ways are disabled. These attributes are the same when all cache ways are disabled as when all ways are enabled.
- **LICW and LICT:** LICW and LICT instructions that access Cache Ways that have been disabled will return undefined results. However, error checking is disabled for disabled Ways so we will never see a parity or ECC violation in this case
- **Locked Cache Lines:** If cache line locking is configured the user should be aware that if by marking some cache ways as unused leaves a situation where all remaining ways for a given index are locked, when it comes time to replace one of those lines, one will be silently unlocked with no exception notification.

23.2 RAM, ROM, and XLMI Organization

If configured, each instruction RAM or ROM, each data RAM or ROM, and the XLMI port are assigned a physical base address at configuration time. This physical base address must be aligned to a multiple of the size of the RAM, ROM, or XLMI port. The processor uses these base addresses to detect when a specific access falls within a memory space of one of the local memories, or the XLMI port.

There can be no address overlap among the local memories or the XLMI port.

As described earlier, the Xtensa LX7 processor can have one or two load/store units. Each load/store unit can independently perform loads or stores. Each data RAM and data ROM port must be connected to all configured load/store units (XLMI cannot be configured with two load/store units). The base addresses of the local memories are the same for all configured load/store units. A C-Box is optionally available for the data RAM interfaces, arbitrates the two load/store unit ports to each data RAM. Data memories connected to two load/store units without a C-Box must be able to deal with access requests from both load/store units by using multi-ported memories or an external memory-port arbiter.

23.3 Memory Combinations, Initialization, and Sharing

23.3.1 Combinations of Cache, RAM, ROM, and the XLMI port

The Xtensa processor can be configured with a maximum of six local data-memory and six local instruction-memory ports. Local data-memory ports include the data-RAM ports, the data-ROM port, the XLMI port, and ports for data-cache ways. Because each way of the data cache requires a separate interface port (each of the cache's ways is a separate data array), a four-way data cache, for example, consumes four of the six available local data ports and therefore allows configuration of just two additional local data memories.

Xtensa configurations with two load/store units can have one or more data RAMs or data ROMs. Xtensa configurations with two Load-Store Units may also have Data Cache. Two load-store units with data cache require that at least 2 Data Cache ways are configured.

Local instruction-memory interfaces include the instruction-RAM ports, the instruction-ROM port, and one way of an instruction cache. There can be at most two instruction-RAM ports and two data-RAM ports. There can be at most one instruction-ROM port, one data-ROM port, and one XLMI port.

For each cache, the number of interfaces—and the number of data/tag memory array pairs—depends on the configured associativity. The maximum allowed cache associativity is four, therefore, the number of cache interfaces (for the data/tag array pairs) can be any integer from zero to four.

Note: The Xtensa LX7 processor must be configured with at least one local data-memory interface and one local instruction-memory interface.

23.3.2 Instruction RAM Initialization and Sharing

Instruction RAM Initialization Boot Options

There are a number of options available for initializing the Xtensa processor's instruction RAM:

- Perform store operations to instruction RAM: The Xtensa processor can explicitly store data to the instruction RAM using the S32I and S32I.N instructions. This method of initializing the Instruction RAM has the advantage that it does not require the `IRam`*n*`Busy` signal (because the instruction RAM need not be initialized by an external agent), which may improve cycle time. However, initializing Instruction RAM in this way can be very slow, because the processor will replay each instruction that follows a store to the instruction RAM. Also, if initialization code must be executed

uncached (if, for instance, there are no other local memories), then the processor must fetch each store instruction over the PIF, which adds further delay. This method may be appropriate if the instruction RAM will only be initialized infrequently and if the slow initialization time is not otherwise a problem.

- Instruction RAM loading via inbound-PIF requests: The processor can be configured to allow inbound-PIF requests to access instruction RAM. This allows external agents such as DMA controllers and other processors on the PIF to write directly to the local instruction RAM. Inbound-PIF access to instruction RAM can be accomplished at reset by asserting the `RunStall` signal (see Section 22.6 “Run/Stall”). Cycle time may suffer due to extra internal multiplexing of the address, data, and control lines going to the instruction RAMs. Also, this method requires that a PIF be configured. Note that if more than one instruction memory is configured, inbound-PIF store requests issued to an instruction RAM can occur in parallel with the execution of instructions out of one of the other local instruction memories.
- External DMA to local instruction RAM: Using the `IRam`*n*`Busy` signal, external multiplexing, and an external memory arbiter, the designer can create a system that loads instruction RAM using external DMA operations. The following figure illustrates such a configuration.

  The external multiplexing allows the external agent to control the instruction RAM and perform write or read operations. Each time the external agent takes control of the instruction RAM, it must assert `IRam`*n*`Busy` on the next cycle to let the processor know that the Instruction RAM was unavailable. Note that the instruction RAM can be initialized before the processor comes out of reset using this method. Cycle time can be affected by the additional multiplexing on the signals to the instruction RAM and by any extra internal logic required in the processor.

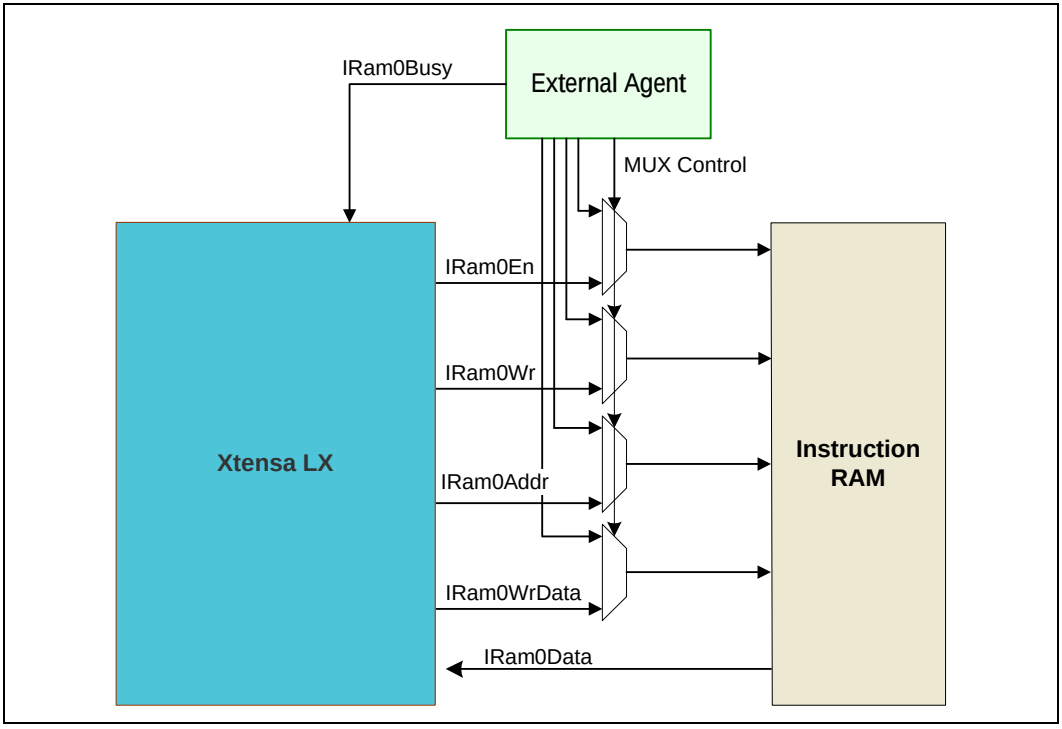


Figure 23–130. Instruction RAM Shared by the Xtensa Core & an External DMA Controller

Sharing an Instruction RAM Between Processors

It is possible to share one instruction RAM between processors. This is not generally advisable for performance reasons. A processor that is executing from the instruction RAM needs access to the RAM most of the time and will stall often waiting to access the shared RAM if a second processor is also using it. Only in certain limited cases will the processor have enough bytes in its internal instruction buffers to avoid the need for frequent instruction-RAM accesses. However, certain applications may find this performance penalty acceptable.

The design for sharing an instruction RAM resembles the design for performing external DMA to local instruction RAM discussed above. An external agent must arbitrate access requests from the two processors and grant instruction-RAM access to one requester. The arbiter may use additional information from each processor to arbitrate instruction-RAM access, such as whether each processor is doing a load or a store or whether each processor is attempting to enable the instruction RAM.

Note: An Xtensa processor connected to other local instruction memories (such as an instruction cache, a second instruction RAM, or an instruction ROM) in addition to an instruction RAM will assert the local instruction-RAM's enable often even though it may not be executing instructions out of that instruction RAM. In many memory-access cases, the Xtensa processor can enable only the local memory that serves as the access target. However, for some cases involving an instruction fetch, the Xtensa processor must access all instruction memories in parallel to determine the appropriate target memory and minimize instruction-fetch latency, thus shifting the time delay required to identify the target memory for the instruction fetch. In these cases, the processor simultaneously performs an instruction fetch from all local instruction memories and then uses the instruction provided by the actual fetch target. Consequently, SOC designs requiring the lowest power dissipation should minimize the number of instruction memories connected to the Xtensa processor.

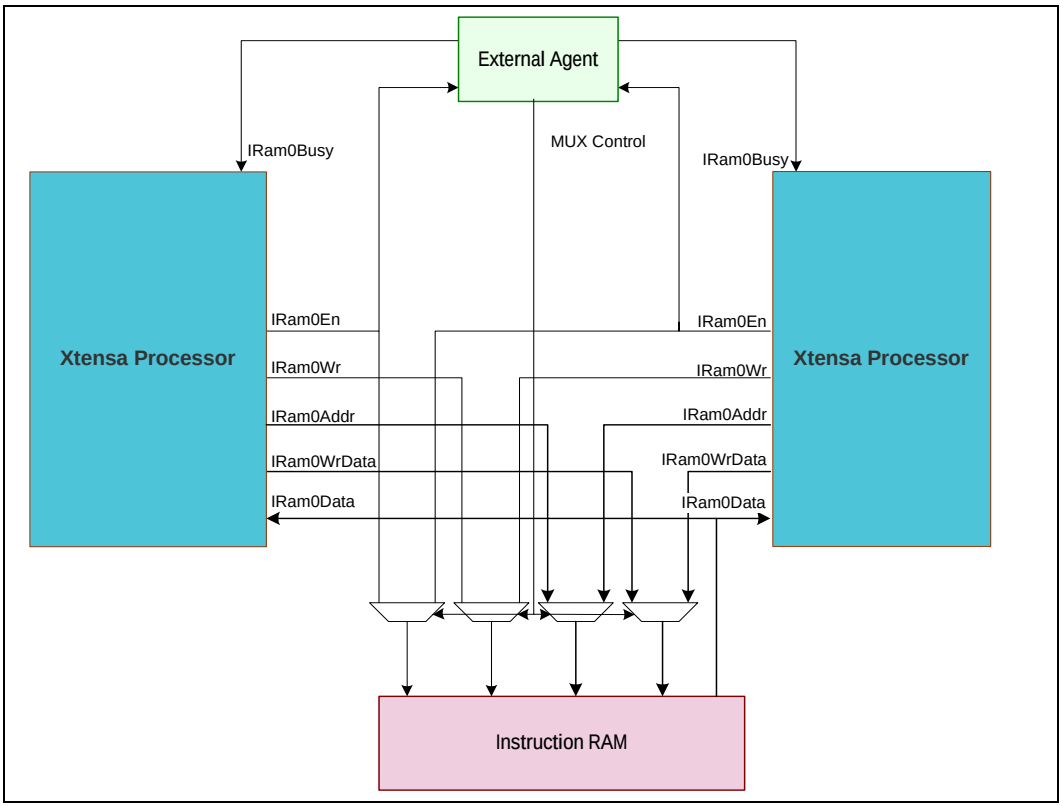


Figure 23–131. Instruction RAM Shared by Two Xtensa Processors

23.3.3 Data RAM and XLMI Initialization and Sharing

Data RAM and XLMI Data Port Initialization Options

The initialization and sharing details below refer to the data RAM, but they apply equally to the XLMI port. There are a number of options available for initializing the processor's data RAM (or to memory connected to the XLMI data port), similar to those available for the instruction RAM:

- Perform store operations to the data RAM: The processor can explicitly store data to the data RAM. This method of initializing the data RAM has the advantage that it does not require the DRam*n*Busy*m* option (because the data RAM need not be initialized by an external agent), which may improve cycle time and yield a processor core that runs at a higher clock frequency. However initializing data RAM in this way can be slow if the initialization data must be read over the PIF. Also, if initialization code must be executed uncached, then the processor must fetch each instruction over the PIF, which adds further delay. This method may be appropriate if the data RAM is initialized infrequently and if slow initialization is not otherwise a problem.
- Data RAM loading via inbound-PIF requests: The processor can be configured to allow inbound-PIF requests to access data RAM. This feature allows external agents such as DMA controllers and other processors connected to the PIF to write directly to the processor's local data RAM. Inbound-PIF access to data RAM can be accomplished at reset by asserting the processor's RunStall signal (see Section 22.6 "Run/Stall"). Cycle time may suffer due to extra internal multiplexing of the address, data, and control lines to the data RAMs. Also, this method requires that the configured processor have a PIF. Note that if more than one data memory is configured, inbound-PIF store requests issued to a data RAM can occur in parallel with data accesses to or from one of the other local data memories.
- External DMA to local data RAM: Using the DRam*n*Busy*m* signal, external multiplexing, and an external memory arbiter, the designer can create a system that loads data RAM using external DMA operations in a way similar to the instruction-RAM example described in the previous section. The external multiplexing allows the external agent to take control of the data RAM and perform read or write operations. Each time the external agent takes control of the data RAM, it must assert DRam*n*Busy*m* on the next cycle to let the processor know that the data RAM was unavailable (see Section 23.3 for details on the use of DRam*n*Busy*m*). Note that the data RAM can be initialized before the processor exits reset using this method.

Sharing a Data RAM or XLMI Between Processors

It is possible to share one data RAM between processors. Performance may suffer if several processors frequently attempt to access the same data RAM at the same time. However, certain applications may find this performance penalty acceptable.

The design for sharing a data RAM resembles the design for sharing local instruction RAM discussed in the previous section. An external agent must arbitrate access requests from the two processors and grant data-RAM access to one requester. The appropriate address and control signals to the shared RAM must be multiplexed and the appropriate DRam*n*Busy*m* signals activated.

See Figure 23–131 for an example of how the multiplexing might be designed.

Note:

- Figure 23–131 shows an example of shared memory using an instruction RAM. The design using a data RAM is the same.
- An Xtensa processor connected to other local data memories (such as a data cache, a second data RAM, or a data ROM) in addition to a data RAM will assert the local data-RAM's enable often even though it may not be accessing data out of that specific RAM. The Xtensa processor accesses all data memories in parallel to determine the appropriate target memory and minimize data-fetch. In these cases, the processor simultaneously performs a fetch from all local data memories, and then uses the data provided by the actual fetch target. Consequently, SOC designs requiring the lowest power dissipation should minimize the number of data memories connected to the Xtensa processor.
- Any shared data memory must correctly support atomic S32C1I operations (see Section 4.6.2 and Section 4.6.3 for this configurable option) without allowing deadlock conditions to arise.
- If either Xtensa LX7 processor has more than one load/store element and the C-Box option is not selected, then each load/store element has its own interface (including the associated DRam*n*Busy*m* signal), so that arbitration and multiplexing must be performed external to the processor for each interface port. Because each load/store element interface is uniform and independent, the circuitry required to share a data RAM between two single load/store element processors is identical to that required to share a data RAM between two load/store elements for the same processor. (For more information on sharing memory, see Appendix B "Notes on Connecting Memory to the Xtensa Core".)

23.4 Memory Transaction Ordering

In certain situations, the load/store unit hardware will (while maintaining data coherence) change the order of reads from and writes to local memories from the corresponding program ordering of the loads and stores. This is distinct from—and subsequent to—any instruction reordering performed by the compiler on the source program.

Within the loads and stores directed at a given memory, the Xtensa processor is *Processor Ordered,* which means that writes are executed in program order but reads can bypass writes as long as they don't bypass writes to the same address as the read target. Reads are also executed in program order with respect to other reads.

Across multiple load/store elements, the Xtensa LX7 processor is *Release Ordered* but not *Processor Ordered*, which means that programmers can explicitly order blocks of memory accesses using the `L32AI`, `S32RI`, or `MEMW` instructions, but the accesses made by the two load/store units are not otherwise ordered. A compiler flag can force Xtensa LX7 programs to be processor ordered by reducing usage of both load/store units at the possible expense of some performance.

Note: Memory accesses to overlapping addresses from two different slots in a FLIX instruction must not both be stores, or a `LoadStoreErrorCause` exception will result. In the case of two overlapping addresses from a FLIX bundle consisting of a load and a store, the load will return the previous data and NOT the updated data from the store.

Across multiple load/store units, the hardware is not processor ordered. However, for a given address, the processor maintains data coherence. To achieve this coherence, the processor stalls a younger store in the pipeline when an older store initiated by a different load/store unit is not yet written to memory or to the processor's write buffer. The stall occurs even if a store operation to the address is disabled through user-defined byte disables (see Section A.9 "GlobalStall Causes" on page 685. Therefore, stores to overlapping addresses initiated by different instruction slots of different FLIX instructions located close to each other in a program may cause some performance degradation. However, there is no performance degradation for load-store or load-load access combinations made to overlapping addresses from different instruction slots of different FLIX instructions.

The processor will replay a younger load operation when that load is directed at an address that matches two older stores that are not yet written to memory or to the processor's write buffer (each store operation having been initiated by one of the processor's two load/store units). The replay of the load forces the load operation to complete after the store operations have completed, thus maintaining data coherence. This load-replay mechanism does account for user-defined byte disables.

23.5 Memory-Access Restrictions and Exceptions

The local memory interfaces are optimized for either data memories or instruction memories. Instruction memory interfaces include the instruction RAM and ROM ports. The instruction memory interface is designed for fast instruction fetches, but is not optimized for load or store operations. With the exception of L32R instructions in configurations without an MMU with a Translation Look Aside Buffer (TLB) and Autorefill, if a 32-bit

load operation accesses an instruction memory, the Xtensa processor will flush its pipeline and will replay the load. If a 32-bit store operation accesses an instruction memory, the processor will flush its pipeline and replay the instruction following the store.

In configurations that do not have an MMU with Translation Look Aside Buffer (TLB) and Autorefill, L32R instructions going to RAM or ROM will execute without flushing the pipeline and replaying, but may cause a single cycle pipeline stall if the instruction fetch logic is also trying to access the memory on the same cycle. This allows firmware literals to be placed in IRAM or IROM with (at most) a small penalty. For best performance however, firmware literals should be placed in data memory, not instruction memory.

In configurations that have an MMU with a TLB and Autorefill logic, even L32R instructions will cause a pipeline flush and replay. Consequently, firmware literals should be placed in data memory, not instruction memory, to avoid significant performance reduction.

Notes:

- When configuring the processor's memory map, care should be taken to assure that the data-memory addresses are located close to the instruction memory addresses to provide fast software access to these literals.
- An 8- or 16-bit load or a store operation directed at an instruction memory will generate a load-store error exception.
- The Xtensa processor's data-memory interfaces include the data ROM, data RAM, and the XLMI ports. Attempting to fetch instructions from memories connected to these ports causes an instruction-fetch error exception.
- Attempting to perform store operations to an instruction ROM or data ROM causes a load-store exception.

These restrictions are summarized in Table 21–93 on page 401.

24. TIE for Hardware Developers

24.1 Processors as Programmable Alternatives to RTL

Most System-On-a-Chip (SOC) designs consist of one or more programmable processors, multiple memory cores, and numerous RTL modules. In a typical design, the performance-critical functions are implemented in RTL, whereas control-oriented functions that need more flexibility are coded in software running on a programmable processor. In many situations, designers need both the performance of custom RTL design and the flexibility of a programmable solution. Creating application-specific extensions to Xtensa microprocessor cores using the TIE language may provide a very attractive solution for such designs.

The TIE language allows you to create new register files, computational units, and custom data paths that significantly enhance the Xtensa processor's computational capability. The Xtensa processor offers the traditional memory and bus interfaces offered by most processors, but TIE language allows you to create new, custom interfaces between the processor and other pieces of SOC hardware bandwidth. The TIE design methodology allows you to develop solutions to design problems using correct-by-construction design techniques that have both the flexibility of a programmable processor and the performance of custom RTL implementation.

This design methodology thus allows you to use processors in many design situations where custom RTL design was previously the only practical approach. The advantage of this approach is that, by using a configurable processor and the TIE language, you can develop these solutions using a pre-verified IP core as a foundation and add custom extensions through correct-by-construction design techniques. This design approach significantly reduces the need for the long verification times required when designing custom RTL.

The following sections provide an overview of some of the major capabilities of the TIE language. Detailed usage of the TIE language can be found in the *Tensilica Instruction Extension (TIE) Language Reference Manual* and *Tensilica Instruction Extension (TIE) Language User's Guide*.

24.2 A Simple Hardware Extension

Consider a simple computation that calculates the average of two 16-bit numbers. On most general-purpose programmable processors, this computation can be performed by executing two instructions: an "add" and a "shift right" of the sum by 1. For a hardware designer implementing such an operation in RTL, it is easy to see that the "shift right by 1" is essentially a free operation (it requires no gates, only wires), and thus the entire

computation can be performed in about the same amount of time as the addition, and with the same number of gates. Using TIE, you can create a new instruction (a hardware execution unit) that performs this computation as follows:

```
operation AVERAGE {out AR res, in AR input0, in AR input1} {} {
     wire [16:0] tmp = input0[15:0] + input1[15:0];
     assign res = tmp[16:1];
}
```

The above instruction takes two input values (`input0` and `input1`) from the Xtensa processor's AR register file to compute its output, `res`. This result is then put back into the AR register file. The semantics of the instruction, an add followed by a right shift, are described using standard Verilog-like syntax. This 3-line description is all you need to create a new processor instruction that computes the average of two 16-bit numbers. From this description, the TIE compiler automatically generates the data path, pipeline and control hardware necessary for the processor to execute this instruction.

24.3 Creating New Registers

The above example illustrates an instruction that operates on values stored in the Xtensa processor's AR register file, which is a 32-bit-wide register file. Now consider the problem of implementing a multiply/accumulate operation that multiplies two 16-bit numbers to generate a 32-bit product, which must be accumulated in a 40-bit accumulator. You need a 40-bit register for the accumulator, but most general-purpose 32-bit processors lack such wide registers. TIE allows you to create new registers (or "state" as they are referred to in the TIE language) that can then be used to store input or output operands of designer-defined instructions. The following piece of code implements the multiply-accumulate instruction as described above.

```
state ACCUM 40 add_read_write

operation MAC {in AR m0, in AR m1} {inout ACCUM} {
     assign ACCUM = TIEmac(m0[15:0], m1[15:0], ACCUM, 1'b1, 1'b0);
}
```

The first line of code declares a new 40-bit state (or register) named `ACCUM`. The `add_read_write` flag is optional and, if present, generates instructions that move data between the AR register file and the new state register. These instructions can be used to initialize the new accumulator, or to copy the accumulator contents into an AR register.

The `MAC` instruction takes two inputs (`m0` and `m1`) from the AR register file and generates an output that is stored in the `ACCUM` state register. TIEmac is a built-in TIE module that takes five arguments. The first two are the multiplier and multiplicand, respectively. The third argument is the accumulator. The fourth argument is a 1-bit flag that indicates whether the multiplication is signed or unsigned, and the fifth argument is a flag that

indicates whether the product is to be added to the accumulator, or subtracted from it. Thus, TIEmac provides a convenient, short-hand notation for specifying a multiply/accumulate operation with different options.

24.4 Multi-Cycle Instructions

The MAC example above implements data path hardware that performs 16-bit multiplication with 40-bit accumulation. For designs that have an aggressive clock-frequency goal, it is possible that this computation may not complete in one processor clock cycle because multipliers are generally slower than other sorts of logic constructs used in processor pipeline stages. To prevent slower operations from degrading the processor's maximum clock speed, the TIE language allows you to spread computations over multiple clock cycles using the *schedule* construct.

The schedule construct specifies the pipeline stage in which the input operands of the computation are used (*use*) and output operands defined (*def*). Instructions with all operands having the same use/def stage are single-cycle instructions. Their input operands are available to the processor's data path logic at the beginning of the specified cycle, and their result is expected by the end of the same cycle. Instructions with one or more output operands that have a def stage later than one or more of the input operands are multi-cycle instructions. For example, the following schedule for the MAC instruction makes it a 2-cycle instruction:

```
schedule sch_mac {MAC} {
     use m0 1;
     use m1 1;
     use ACCUM 2;
     def ACCUM 2;
}
```

The above schedule specifies that the multiplier (`m0`) and multiplicand (`m1`) are used in stage 1, whereas the accumulator (`ACCUM`) is used and defined in stage 2. The hardware thus has two clock cycles to perform the computation, which should be an adequate amount of time for most SOC designs.

From the operation and schedule description of the MAC instruction, the TIE compiler implements all the hardware necessary for the multiply/accumulate data path, as well as control logic to interface this database to the processor's pipeline. The data path is implemented in a fully pipelined manner, which means that it's possible to issue a MAC instruction every cycle even though it requires two cycles for it to compute the result.

The TIE compiler allows the designer more control over multi-cycle TIE instruction implementations. This enhancement to the TIE compiler may alleviate the need for behavioral retiming by allowing the designer to schedule wire usage.

24.5 Wide Data Paths and Execution Units

The base Xtensa processor contains a 32-bit register file and a 32-bit memory interface. Many computational tasks require processing data whose native size is not 32-bits. Some tasks require a wider memory interface because of the sheer bandwidth requirement of the application. Processing such data on generic 32-bit processors is cumbersome and time consuming, because each native data transfer and computation must be decomposed into 32-bit chunks.

The TIE language allows you to create wide register files and a wide memory interface. It also allows you to create execution units that directly operate on the values stored in these wide register files.

Consider the problem of processing 8-bit video or pixel data. Assume that you need to process eight pixels in one cycle to meet your overall system-throughput goals. An example piece of TIE code for designing such hardware is as follows:

```
regfile VR 64 8 v

operation VADD {out VR sum, in VR in0, in VR in1} {} {
     wire [7:0] sum0 = in0[ 7: 0] + in1[ 7: 0];
     wire [7:0] sum1 = in0[15: 8] + in1[15: 8];
     wire [7:0] sum2 = in0[23:16] + in1[23:16];
     wire [7:0] sum3 = in0[31:24] + in1[31:24];
     wire [7:0] sum4 = in0[39:32] + in1[39:32];
     wire [7:0] sum5 = in0[47:40] + in1[47:40];
     wire [7:0] sum6 = in0[55:48] + in1[55:48];
     wire [7:0] sum7 = in0[63:56] + in1[63:56];

     assign sum = {sum7, sum6, sum5, sum4, sum3, sum2, sum1, sum0};
}
```

The regfile construct defines a new register file that is 64 bits wide and has eight entries. Each of these entries can be viewed as a vector consisting of eight 8-bit pixels. The VADD operation creates an execution unit that performs eight 8-bit additions in parallel. Additional vector instructions can be defined in a similar manner to create a vector-processing engine that is capable of performing a variety of operations on eight pixels in one instruction. Such machines are commonly referred to as Single Instruction Multiple Data (SIMD) machines.

Because the VR register file is 64 bits wide, it is also appropriate to configure the Xtensa processor to have a 64-bit-wide memory interface. This wider memory interface lets the processor load and store entire 64-bit vectors in one memory access, thus doubling the memory bandwidth compared to a standard 32-bit memory interface. Note that the Xtensa processor's memory interface can be configured to be 32, 64, 128, 256, or 512 bits wide.

Xplorer's Vectorization Assistant guides the programmer to the best performance by directing them to areas in the source code that prevent vectorization along with hints as to how to change that.

24.6 Sharing Hardware Between Execution Units

In all the TIE examples so far, we have created one new processor instruction, which the TIE compiler implements as one new hardware execution unit. By default, the TIE compiler generates a separate hardware execution unit for each new designer-defined TIE instruction. However, there are many situations in which a particular piece of hardware can be shared between multiple execution units. For example, one adder can be used to compute the sum or the difference of two numbers and can be used for several custom addition and subtraction operations. “Multiply-add” and “multiply-subtract” computations can share a multiplier. The TIE language allows you to share hardware between multiple execution units through the use of `semantic` sections.

Consider the ADD16 and SUB16 instructions as defined below. These instructions perform addition and subtraction operations on the AR register file, but treat each 32-bit register-file entry as two independent 16-bit numbers.

```
operation ADD16 {out AR res, in AR in0, in AR in1} {} {

     wire [15:0] sum0 = in0[15: 0] + in1[15: 0];
     wire [15:0] sum1 = in0[31:16] + in1[31:16];

     assign res = {sum1, sum0};
}
operation SUB16 {out AR res, in AR in0, in AR in1} {} {

     wire [15:0] diff0 = in0[15: 0] - in1[15: 0];
     wire [15:0] diff1 = in0[31:16] - in1[31:16];

     assign res = {diff1, diff0};
}
```

By default, the processor generator and TIE compiler implement these instructions using independent hardware execution units. However, by defining a TIE *semantic*, these two instructions can be implemented with a single execution unit that shares the adders required for the computation:

```
semantic addsub16 {ADD16, SUB16} {

     wire [31:0] in1_op = ADD16 ? in1 : ~in1;
     wire carry_in = SUB16;
```

```
        wire [15:0] res0 = TIEadd(in0[15: 0], in1_op[15: 0], carry_in);
        wire [15:0] res1 = TIEadd(in0[31:16], in1_op[31:16], carry_in);

        assign res = {res1, res0}
}
```

There are several points to note about the semantic description above. This semantic, named addsub16, implements the two instructions: ADD16 and SUB16. It uses the built-in TIE module TIEadd, which provides an efficient implementation of an addition with a "carry in". In this description, the subtraction operation is implemented by taking the two's complement value of input operand in1 and then adding it to operand in0. Inside semantic addsub16, ADD16, and SUB16 are single-bit, 1-hot-encoded signals that are true when the respective instruction is being executed. These signals are used to choose the appropriate operands for the TIEadd module so that it performs an addition or subtraction depending upon the instruction being executed.

Any number of instructions can be combined into one TIE semantic description using the technique illustrated above. A semantic description that implements many different instructions, all of which share a particular piece of hardware, is likely to become complex and difficult to read. The TIE language provides the *function* construct that makes semantic descriptions easier to write. This construct is described in the next section.

24.7 TIE Functions

A TIE function is similar to a function in Verilog. It encapsulates some computation, which can then be used in different places in a TIE description. TIE functions can be used to make the code modular, by providing one description of a computation that gets instantiated in multiple places. It can also be used to share hardware between multiple semantic blocks by declaring the function to be shared. The following piece of TIE code provides an alternative implementation of the ADD16/SUB16 example of Section 24.5:

```
function [15:0] as16 ([15:0] a, [15:0] b, add) {
        wire [15:0] tmp = add ? b : ~b;
        wire carry_in = add ? 1'b0 : 1'b1;

        assign as16 = TIEadd(a, tmp, carry_in);
}
function [15:0] addsub0 ([15:0] a, [15:0] b, add) shared {
        assign addsub0 = as16(a, b, add);
}
function [15:0] addsub1 ([15:0] a, [15:0] b, add) shared {
        assign addsub1 = as16(a, b, add);
}
```

```
semantic add16 {ADD16} {
     wire [15:0] sum0 = addsub0(in0[15: 0], in1[15: 0], 1'b1);
     wire [15:0] sum1 = addsub1(in0[31:16], in1[31:16], 1'b1);

     assign res = {sum1, sum0};
}
semantic sub16 {SUB16} {
     wire [15:0] diff0 = addsub0(in0[15: 0], in1[15: 0], 1'b0);
     wire [15:0] diff1 = addsub1(in0[31:16], in1[31:16], 1'b0);

     assign res = {diff1, diff0};
}
```

In this example, `as16` is a function that performs a 16-bit addition or subtraction based on the 1-bit function argument `add`. The primary purpose of this function is to encapsulate the hardware that performs the addition and subtraction. Every instantiation of this function will result in a new and unique copy of the hardware that implements the function.

Functions `addsub0` and `addsub1` are defined as shared functions. There is only one instance of the hardware corresponding to this function and it is shared by all instantiations of the function. These functions are used in the semantic description for both the `ADD16` and `SUB16` instructions. The TIE compiler automatically generates the data path and control logic to share the `addsub0` and `addsub1` hardware between the `ADD16` and `SUB16` instructions.

The semantic descriptions of this section look very similar to the operation descriptions in Section 24.5. Hardware sharing is expressed through the use of shared functions instead of writing a common semantic for the two instructions and explicitly specifying the operand multiplexing inside this semantic. Thus, shared functions simplify the specification of instructions that share hardware.

24.8 FLIX - Flexible-Length Instruction Xtensions

FLIX (Flexible-Length Instruction Xtensions) is a powerful TIE feature that allows you to create multiple execution units that operate in parallel and instruction formats with multiple opcode slots. Each opcode slot can hold a combination of existing Xtensa LX7 instructions and new, designer-defined instructions. FLIX instruction formats can be any number of bits from 32-bits wide to 128-bits wide in 8-bit increments.

Long instructions allow more encoding freedom, where a large number of sub-instruction or operation slots can be defined (although three to six independent slots are typical) depending on the operational richness required in each slot. The operation slots need not be equally sized. Big slots (20-30 bits) accommodate a wide variety of opcodes, relatively deep register files (16-32 entries), and three or four register-operand specifiers. Developers should consider creating processors with big operation slots for applications with modest degrees of parallelism but a strong need for flexibility and generality within the application domain.

Small slots (8-16 bits) lend themselves to direct specification of movement among small register sets and allow a large number of independent slots to be packed into a long instruction word. Each of the larger number of slots offers a more limited range of operations, fewer specifiers (or more implied operands), and shallower register files. Developers should consider creating processors with many small slots for applications with a high degree parallelism among many specialized function units.

The following TIE code is a short but complete example of a very simple long-instruction word processor described in TIE with FLIX technology. It relies entirely on built-in definitions of 32-bit integer operations, and defines no new operations. It creates a processor with a high degree of potential parallelism even for applications written purely in terms of standard C integer operations and data-types. The first of three slots supports all the commonly used integer operations, including ALU operations, loads, stores, jumps and branches. The second slot offers loads and stores, plus the most common ALU operations. The third slot offers a full complement of ALU operations, but no loads and stores.

```
format format1 64 {base_slot, ldst_slot, alu_slot}

slot_opcodes base_slot {ADD.N, ADDX2, ADDX4, SUB, SUBX2, SUBX4,
ADDI.N, AND, OR, XOR, BEQZ.N, BNEZ.N, BGEZ, BEQI, BNEI, BGEI, BNEI,
BLTI, BEQ, BNE, BGE, BLT, BGEU, BLTU, L32I.N, L32R, L16UI, L16SI, L8UI,
S32I.N, S16I, S8I, SLLI, SRLI, SRAI, J, JX, MOVI.N }

slot_opcodes ldst_slot { ADD.N, SUB, ADDI.N, L32I.N, L32R, L16UI,
L16SI, L8UI, S32I.N, S16I, S8I, MOVI.N }

slot_opcodes alu_slot {ADD.N, ADDX2, ADDX4, SUB, SUBX2, SUBX4, ADDI.N,
AND, OR, XOR, SLLI, SRLI, SRAI, MOVI.N }
```

The first line of the example declares a new 64-bit-wide FLIX instruction format containing three operation slots: the base_slot, the ldst_slot, and the alu_slot. The second line lists all the instructions that can be packed into the first of those slots: base_slot. In this example, all the instructions happen to be pre-defined Xtensa LX7 instructions. However, new TIE instructions can also be included in this slot.

The processor generator automatically creates a NOP (no operation) for each FLIX operation slot, so the software tools can always create complete FLIX instructions, even when no other operations for an operation slot are available for packing into a long instruction. The last two lines of the example designate the subset of instructions that can go into the other two operation slots.

Thus FLIX allows the generation of multiple execution units that can operate in parallel, resulting in higher computational throughput. FLIX technology may be especially useful when using the Xtensa LX7 processor to implement algorithms and tasks that previously could only be performed by hard-wired RTL blocks.

24.8.1 FLIX3 Option

The FLIX3 option has a 32-bit data path with the base 24/16-bit Xtensa ISA and 64-bit FLIX instructions, as described in Section 9.3.

24.9 Sharing Hardware Among FLIX Slots

Operations in different slots of the same instruction format are executed in parallel. However, operations in different slots of different instruction formats are not executed in parallel. Thus, if an operation appears in two slots of different formats, their hardware can be shared without sacrificing the software performance. This is illustrated in the following example:

```
format fmt 64 {slot0, slot1}
slot_opcodes slot0 {L32I.N, L32R, L16UI, L16SI, L8UI, S32I.N, S16I,
S8I}
slot_opcodes slot1 {foo}
slot_opcodes Inst {foo}

operation foo {out AR a, in AR b, in AR c} {} {
     assign a = b + (c << 1);
}
```

In this example, load store operations such as `L32I.N`, `L32R` etc. appear in both `Inst` and `slot0` slots. Because they both appear in the index zero of the 24-bit and 64-bit formats, their hardware are shared automatically.

Operation `foo` appears in both `Inst` and `slot1` slots. They appear in index zero of the 24-bit format and index one of the 64-bit format. By default the TIE compiler instantiates two copies of the operation `foo`, one for slot index zero and one for slot index one. However, because the operation `foo` is not referenced in two slots of the same format, at most one instance of the operation foo is active in any given cycle. Thus, it is possible to generate one instance of the operation `foo`. You can specify the following TIE property to instruct the TIE compile to generate one instance:

```
property shared_semantic foo {0, 1}
```

The FLIX technology together with the hardware sharing techniques provide the designers great freedom in balancing the software performance and the hardware cost.

24.10 Designer-Defined TIE Interfaces

General-purpose microprocessors communicate with external devices using a set of interface signals collectively referred to as a bus. A typical bus interface allows the microprocessor to communicate with other devices using a “request/response” protocol. Using a microprocessor for high-speed control or data processing in an SOC requires a more flexible communication model. The TIE language allows you to create direct interfaces to other SOC blocks through the use of TIE ports, as illustrated below.

24.10.1 TIE Ports

As explained earlier, you can create additional registers using the TIE state construct. Each designer-defined state can also be “exported” outside of the Xtensa LX7 processor using the export keyword in the state declaration. Exporting the state creates a new, dedicated output port. This output port can then be connected to other components on the SOC that need quick access to the value of the exported state.

The TIE language also supports an `import_wire` construct that defines a new input port to the Xtensa LX7 processor. This input port directly transports values into the processor and these values can be directly used by designer-defined TIE instructions. An `import_wire` is declared and used as follows:

```
import_wire EXT_STATUS 4

operation READ_EXTSTAT {out AR stat} {in EXT_STATUS} {
     assign stat = EXT_STATUS;
}
```

The `import_wire` declaration above creates a 4-bit input bus named `TIE_EXT_STATUS` at the boundary of the Xtensa core that can be connected to any other SOC block in the system. This bus is connected to the SOC block that provides this information to the

Xtensa LX7 processor. Inside the processor the "TIE_" prefix is dropped and EXT_STATUS can be used directly as an input operand to any designer-defined TIE instruction as illustrated with the instruction READ_EXTSTAT. This instruction reads the value of EXT_STATUS and assigns it to an Xtensa LX7 core AR register. Subsequently, any of the Xtensa core instructions can be used to process this value. Note that the imported state EXT_STATUS can also be directly used as an input operand in a computation without first moving it into an intermediate state register or a register file.

Reads from an `import_wire` should not have any effect on the other system blocks, that is, this wire (or bus) can be read multiple times by instructions executed by the Xtensa LX7 processor without creating side effects. The Xtensa LX7 processor generates no output that indicates to external logic when the imported wire (or bus) is being read. The `import_wire` construct is often used to poll the status of a device or of another processor on the SOC.

24.10.2 TIE Queue Interfaces

A third type of designer-defined port, in addition to the exported state and imported wire, is the queue-interface construct, which automatically masks the speculative nature of the `import_wire` construct. A queue is commonly used in hardware design to communicate between two modules. The module that generates the data *pushes* data onto the queue, and the module that consumes the data *pops* data from the queue at a later time when it is ready to process that data. The TIE language makes it easy to create new interfaces that communicate with input and output FIFO queues. *(Note: The TIE queue construct will generate an interface port for a FIFO queue, but it does not create the FIFO queue itself, which is external to the processor.)*

The following example describes a TIE instruction that reads from an input queue, adds a value to it, and writes the result to an output queue.

```
queue InQ 32 in
queue OutQ 32 out

operation MQ {in AR val} {in InQ, out OutQ} {
     assign OutQ = InQ + val;
}
```

The first two statements declare an input queue named `InQ` that is 32 bits wide and an output queue named `OutQ` that is also 32 bits wide. Input queues can be used like any other input operand to an instruction and output queues can be assigned values just like any other output operand.

The TIE compiler translates the queue declaration above into a set of interface signals at the boundary of the Xtensa LX7 processor. The input queue declaration translates to an input signal named `TIE_InQ_Empty`, a 32-bit input bus named `TIE_InQ`, and an

output signal named TIE_InQ_PopReq. Note that the actual hardware queue is outside the Xtensa LX7 processor and these signals interface to the “read” end of the hardware queue.

In the simplest case, the Xtensa LX7 processor asserts the TIE_InQ_PopReq following the execution of an MQ instruction, indicating its desire to read from the input queue. The data on the bus TIE_InQ is read, assuming that there is data in the queue to be read (TIE_InQ_Empty is false). If the input queue is asserting its empty signal (indicating that the input queue is empty), the MQ instruction will stall until the queue is no longer empty.

Complications to this simple case arise when the MQ instruction is killed in the processor’s pipeline (by an exception, for example) because processor reads from TIE queues and TIE ports are speculative. If the MQ instruction reaches the pipeline’s E stage before it’s killed, the processor will initiate the input-queue read operation. If the MQ instruction is subsequently killed, then the data obtained from the subsequently completed input-queue operation will be stored in a buffer inside the Xtensa LX7 processor and is used by the next MQ instruction that reads from the queue.

The TIE compiler translates the output queue declaration into an input signal named TIE_OutQ_Full, a 32-bit output bus named TIE_QutQ, and an output signal named TIE_OutQ_PushReq. When the MQ instruction executes, it makes the result of the computation available on the TIE_OutQ data bus, and asserts TIE_OutQ_PushReq. If the TIE_OutQ_Full signal is deasserted, the data is assumed to have been written to the queue. On the other hand, the processor stalls if the output queue is full (the TIE_OutQ_Full signal is true). Output queues are only written after the instruction has gone past the commit stage of the processor pipeline, so queue-write operations are never killed.

The TIE compiler automatically generates all of the logic necessary to generate the control signals that interface to the queue, the logic to detect the stalls, and the buffering of the speculative reads in the case of an input queue, based on the simple declaration of an input or output queue. TIE ports and queues can be simulated using the Xtensa Instruction-set simulator (ISS) in the XTMP simulation environment and the XTSC (Xtensa SystemC) package.

24.10.3 TIE Lookups

A fourth type of designer-define port is the lookup construct, which is used to perform table "lookups". Some constant tables can be statically defined within the processor using the TIE table construct, while other tables are either more dynamic or require larger data arrays than can be efficiently implemented with synthesizable logic. Tables with dynamic updates can be implemented in memory, but are then limited to the predefined number of memory interfaces, allowed data widths, and fixed pipeline stages. The TIE language

makes it easy to overcome these limitations of standard memory interfaces by introducing the lookup construct. Lookup's can be defined with the exact index (similar to an address), data widths, and latencies that a particular table lookup demands.

The following example describes a TIE instruction that performs a lookup on an interface called MYTABLE. This interface has a 6-bit index which corresponds to 64-entries, and a 256-bit wide entry. The designer-defined operation LKUP performs a "lookup" of MYTABLE and writes the results into a 256-bit wide state, which can be used by other operations.

```
lookup MYTABLE {6, Estage} {256, Estage+1}
state VAL 256 add_read_write
operation LKUP {in AR index} {out MYTABLE_Out, in MYTABLE_In, out VAL}
{
     assign MYTABLE_Out = index;
     assign VAL         = MYTABLE_In;
}
```

TIE lookups are not strictly limited to simple table lookups, although table lookups are an ideal application for TIE lookups. The lookup can be generically thought of as having a request phase with a fixed-latency response, and an optional request arbitration. While TIE lookups may resemble other memory interfaces, it is important to distinguish TIE lookups from memory interfaces. Most notably, no facility to store data to a TIE lookup interface is provided, and the accompanying software tools will not recognize TIE lookup operations as having memory semantics (including memory allocation or alias analysis).

TIE ports, queues and lookups allow the hardware designer to create custom interfaces between modules, in much the same way that such interfaces are created during the design of logic blocks using RTL descriptions. Thus the Xtensa LX7 processor is not limited to the fixed interfaces, interface protocols, and limited bandwidth of traditional processor bus interfaces.

24.11 System Design Using TIE Ports, Queues, and Lookups

Most TIE constructs result in the generation of data path and control logic internal to the Xtensa processor. All the interface signals to and from these TIE modules are automatically connected to the appropriate interfaces within the microprocessor's core. However, TIE ports, TIE queues, and TIE lookups create new, designer-defined interfaces at the processor's boundary. The system designer is then responsible for connecting these interface signals to appropriate modules in their design.

This chapter provides general information on connecting these interfaces correctly. Timing diagrams are also provided to help the system designer understand the behavior of these interface signals.

Chapter 35 lists the AC timing specifications for these interfaces.

24.11.1 TIE Ports: State Export and Import Wires

Designers can create their own top-level processor interfaces with the TIE state export and import wire constructs. These interfaces can be used for many purposes, including multiple-processor synchronization, device control, and status reporting.

TIE State Export

The TIE state export construct allows designer-defined, processor-internal state to be made visible to hardware external to the processor via top-level pins. The exporting of designer-defined state can be a useful system-design tool for efficiently exposing status and control information between multiple processors and between processors and other devices.

The architectural copy (committed state) of an exported state is visible on the top-level pin named `TIE_<name>`, as shown in Figure 24–132.

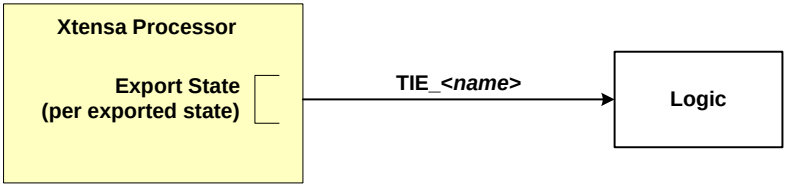


Figure 24–132. Exported State Block Diagram

The architectural copy of a state is updated on any committed write to that state, and this state becomes visible on the associated interface pins in the cycle following the W stage of the instruction that wrote the exported state. Note that if an instruction that writes an exported state is stalled in the W stage (for example, by the `RunStall` signal), then the state update will not be visible until the stall is removed.

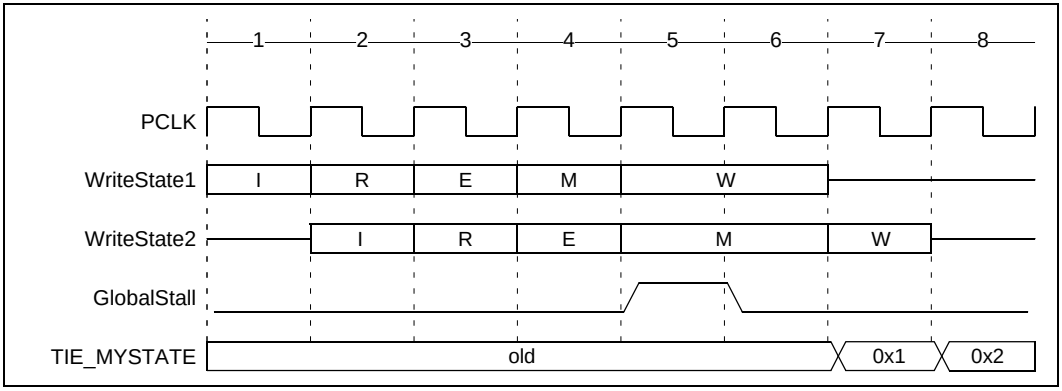


Figure 24–133. State Export Example

In Figure 24–133, the first instruction is stalled in the W stage (cycle 5) for 1-cycle. It updates the architectural copy of the state, `MYSTATE`, in cycle 6 and is visible on the interface, `TIE_MYSTATE`, in cycle 7.

TIE Import Wires

Import wires allow designer-defined TIE instructions to read inputs from a processor core's designer-defined, top-level pins. These interfaces can be connected to external hardware, including other processors and hardware devices. For example, an import wire can be used to read the exported state of another Xtensa processor.

Import wires are automatically registered within the processor before they become available for use in instruction semantics. Registering of import wires allows external logic to have late timing on the input wire, while allowing instruction semantics to have early timing.

Import wires are available for use in the E-stage of TIE instructions. Thus, Tie-import reads are speculative, because the instruction can be killed before it reaches the commit stage in the processor's pipeline. If the instruction is killed, the read value is simply discarded. It is assumed that the value on the import wire is always available to be read; there is no way to stall the processor on the read of an import wire because no "handshake" is involved in the reading of an import wire. The external logic does not receive any acknowledgement when the processor reads the state of the input wire.

24.11.2 A Simple Multiple Processor System Communicating via TIE Ports

In the example system block diagram below, a master processor called Processor 1 can force a slave processor, called Processor 3, into a stalled state via the `RunStall` port with an export state, `TIE_StallProc`. The master processor can then cause another processor, Processor 2 (which is serving as a DMA engine) to initialize the instruction memory of Processor 3.

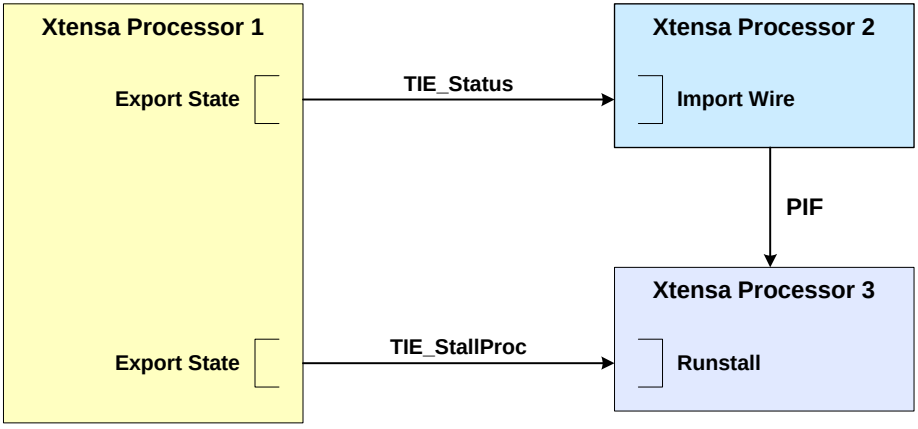


Figure 24–134. Example System Connected through Export State and Import Wires

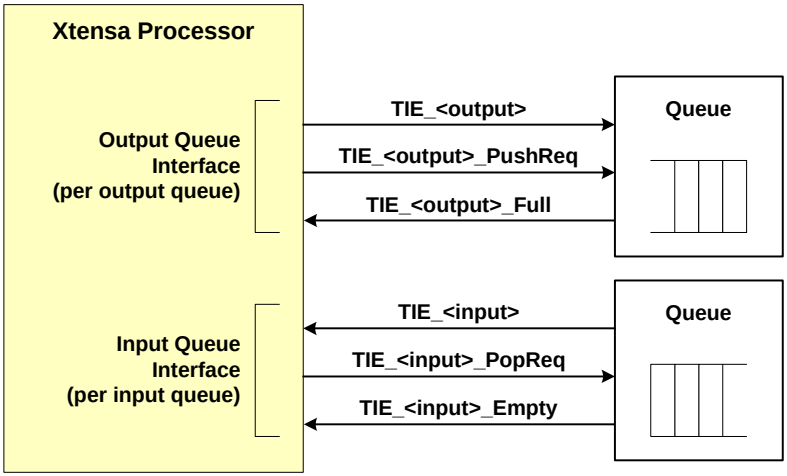
Many other systems and uses of exported states and import wires are possible.

24.11.3 TIE Queue Interfaces

TIE queue interfaces allow Xtensa processors to directly interface to external FIFO queues that have standard push- and pop-type interfaces and full-and-empty flow control. Designer-defined instructions can push data from the processor onto external queues, pop data from external queues into processor state registers or register files, or use the information from input queues directly in computations.

TIE queue interfaces should only be connected to synchronous, First-In, First-Out (FIFO) memory devices. The Xtensa TIE queue interface is synchronous in the sense that the data transfer between the Xtensa processor's queue interface and the external FIFO queue occurs on the rising edge of the processor's internal clock signal.

Figure 24–135 shows the input and output TIE queue interfaces.



<input> and <output> refer to the respective <name> of the queues.

Figure 24–135. Interface Signals for TIE Queues

TIE Input Queue Interfaces

A TIE input queue interface is used to read data from an external FIFO queue. Queue data is available for use in instruction semantics in the processor pipeline's M stage. The queue data is registered within the processor and is therefore read at least one cycle before the data is needed by an instruction in the M stage.

When a TIE input queue interface is defined, a `PopReq`-and-`Empty` handshake signal pair is automatically created along with the queue interface's data lines. The Xtensa processor asserts the associated `PopReq` (pop request or read enable) signal whenever it wants to read data from an input queue. If `PopReq` is asserted and `Empty` is not asserted, the processor latches the data on the queue interface's data lines and the read operation is completed. The external FIFO queue is expected to present valid data on the data interface whenever the empty signal is not asserted.

If the `Empty` signal is also asserted during a cycle when the Xtensa LX7 processor asserts `PopReq`, no data transfer takes place. In this situation, the processor can continue to assert `PopReq` during the next cycle or it can remove the request. The processor may cancel the pop request for a variety of reasons, such as the occurrence of an interrupt. Thus, the TIE input queue's data-transfer protocol is a cycle-by-cycle based protocol. Each cycle is independent of the previous cycle(s).

If the queue's Empty signal is asserted when the processor wants to read data from an input queue and instruction execution cannot continue without the data, the processor's pipeline will stall and wait until the Empty signal is no longer asserted. This behavior is called a *blocking-queue* behavior. Note that the processor can continue to assert its pop-request signal when the external queue is empty. The FIFO queue design should be such that this does not cause the queue to underflow; instead, the FIFO queue must ignore the processor's pop request when the queue is empty.

As with many read operations, the Xtensa processor can perform speculative reads on TIE input queues. The external queue is read before an instruction's commit stage, so there is a possibility that the instruction using the popped queue data may not commit due to a control-flow change or an exception. The processor handles the speculative nature of external queue reads by internally buffering all queue reads. Thus external FIFO queues need not be designed to handle speculative reads. The Xtensa TIE input-queue interface is already designed to handle such situations.

TIE queue reads are always speculative, because an external queue may be popped and the instruction that popped the queue may not commit its write to any architectural state. The processor buffers the queue data internally until the next queue-read instruction executes. That instruction will read the value saved in the processor's internal buffer rather than a new value from the external queue. There can an arbitrary amount of time between when an external queue is popped and when the popped data is consumed by an instruction executing in the processor.

The TIE input queue interface serves as a decoupling mechanism between the external queue and the processor's pipeline. It provides sufficient buffering to allow data to be popped from the external queue every cycle. This approach allows sustained, high-bandwidth data transfers from external FIFO queues.

The following TIE code example and subsequent timing diagrams illustrate the various aspects of an input queue interface. The example instantiates an input queue named QIN, an exported state named LAST_READ, and an instruction that reads the QIN input queue and writes that value to the LAST_READ state.

```
queue QIN 32 in
state LAST_READ 32 32'b0 add_read_write export
operation READQ {} {out LAST_READ, in QIN} [
     assign LAST_READ = QIN;
}
schedule READQ {READQ} { def LAST_READ 2; }
```

Data From Input-Queue Reads Available in M-Stage

Figure 24–136 shows an input-queue read access. In cycle 3, the processor asserts the pop-request signal TIE_QIN_PopReq. The queue contains data (it is not empty), as indicated by the negated signal TIE_QIN_Empty. Consequently, the processor latches the value on the queue data bus TIE_QIN on the rising edge of the clock between cycle 3 and cycle 4. The figure shows that the sampled data is not written speculatively to the TIE_LAST_READ state in cycle 4. Instead, the instruction commits in cycle 5 and the new state value becomes visible on the TIE_LAST_READ port in cycle 6 via the export-state mechanism. Note that the external queue must present valid data on the data bus when Empty is not asserted.

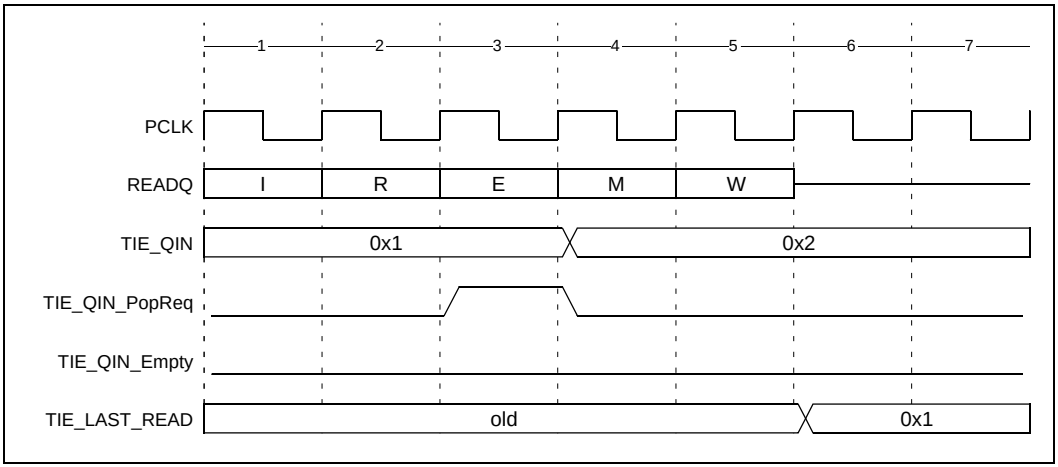


Figure 24–136. TIE Input Queue Read

Input queue data is always available to an instruction semantic in its M stage for both 5-and 7-stage processor pipeline configurations. The pop request is issued to the external FIFO interface at least one cycle before the M-stage for both 5- and 7- stage processor pipelines.

Input Queue Empty Interface

Figure 24–137 shows the same basic sequence as Figure 24–136, except that the queue is empty when the pop request is made. The processor stalls until TIE_QIN_Empty is negated in cycle 4. The read data is latched on the rising edge of the clock between cycle 4 and cycle 5. The queue again becomes empty as a result of this read, so TIE_QIN_Empty is asserted in cycle 5. This diagram also illustrates that the queue data bus, TIE_QIN, need not be valid during any cycle in which TIE_QIN_Empty is asserted.

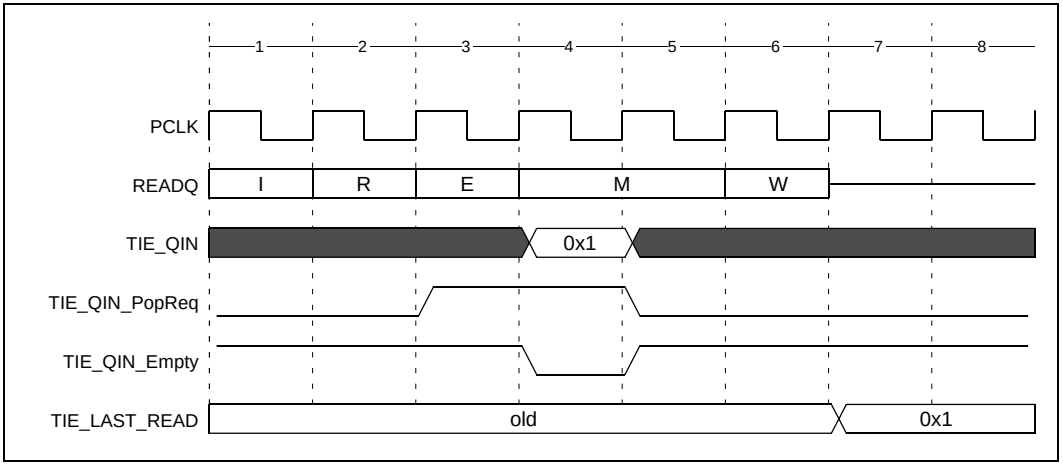


Figure 24–137. TIE Input Queue Read Stalled 1-Cycle Due to Empty

Note: If the input queue remains empty and the stall lasts for sixteen cycles or more, there may be an extra stall cycle when the queue finally becomes non-empty. This extra stall is required in some cases to guarantee that all memory enables can be turned off for power savings in the case of a long queue stall condition, and then turned back on correctly when operation resumes.

Figure 24–138 shows a sequence in which the processor asserts `TIE_QIN_PopReq` to read the input queue, but the queue is empty. Subsequently, the pop request is removed, even though no data transfer took place. The request is not made again, even when the queue is not empty. This diagram illustrates that the external FIFO queue design should not assume that an asserted `TIE_QIN_PopReq` signal will remain asserted until a data transfer occurs. Again, the external FIFO queue's interface should be designed to respond to the input-queue interface signals on a cycle-by-cycle basis.

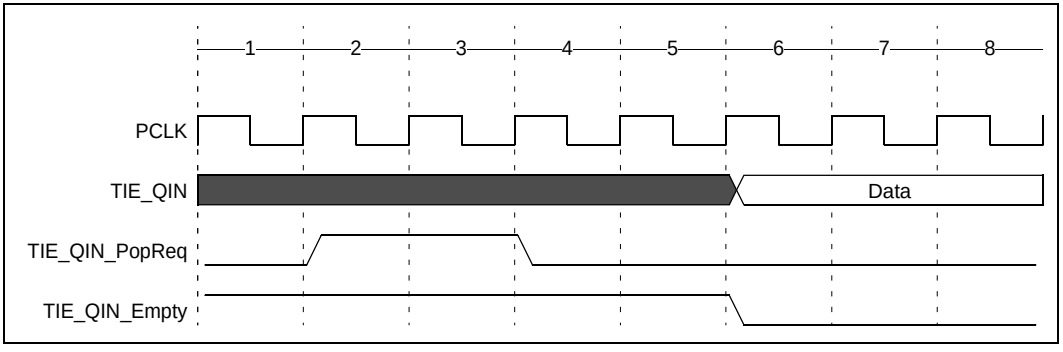


Figure 24–138. TIE Input Queue Read Request Withdrawn

In Figure 24–139, the processor attempts several queue reads, but they are aborted by an interrupt. Both READQ1 and READQ2 pop data from the external queue during cycles 3 and 4, respectively, but an interrupt causes these instructions to be aborted in cycle 5. When the program returns from the interrupt handler and re-issues a queue-read instruction, that instruction obtains the internally buffered queue value previously read during cycle 3 rather than causing the re-assertion of the pop signal in cycle 9, which would then (erroneously) read another piece of data from the external FIFO queue.

Input-queue reads can be aborted for many reasons, including interrupts and exceptions. In all such cases, the processor buffers the input queue data in an internal buffer so that no data is lost. That internal buffer data is not retained if the core is reset or powered down.

Input Queue Speculative Buffering

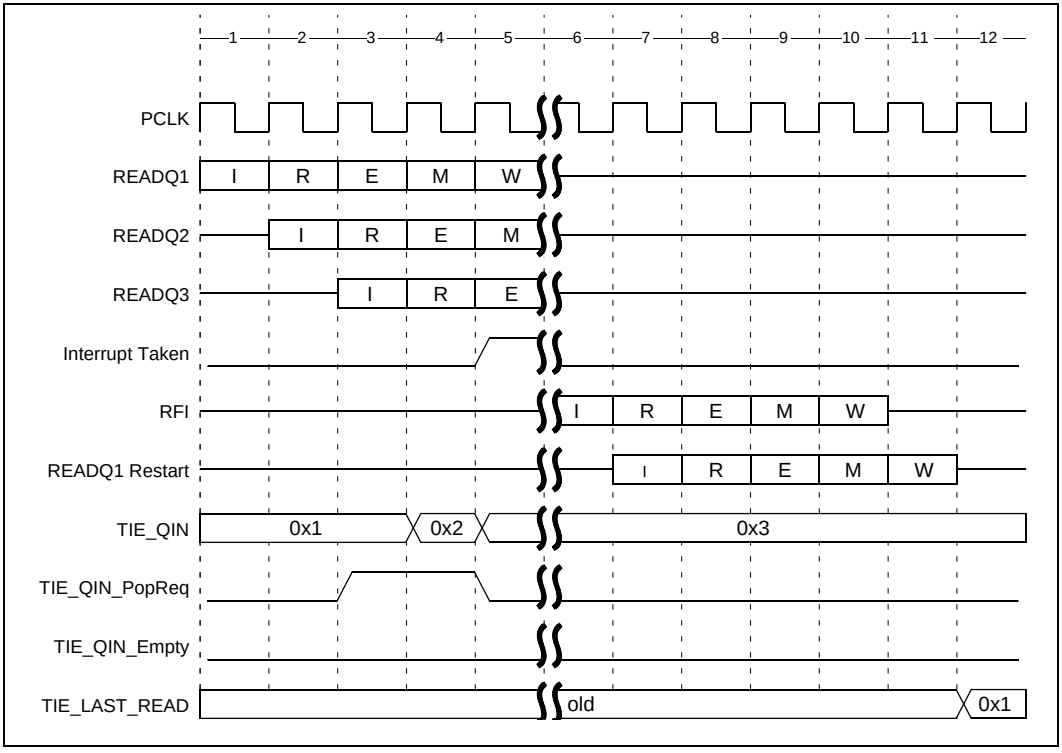


Figure 24–139. TIE Input Queue Read is Killed and Retried Later

TIE Output Queues

The Xtensa processor uses TIE output queues to write data to external FIFO queues. Instructions that write to an output queue interface must have the data available in their M stage. The Xtensa processor always registers the data written to an output queue before it reaches the processor's output pin, which maximizes the amount of time within a clock cycle that logic external to the processor has to operate on the output-queue data. Data is first available to the external FIFO queue in the instruction's W stage. If the instruction writing to the output queue does not commit, the processor will kill the output-queue write before it attempts to write to the external queue interface. Thus, the Xtensa processor never performs speculative writes to external output queue interfaces.

Defining a TIE output queue automatically creates a `PushReq`-and-`Full` handshake signal pair along with the required number of output data lines to the external FIFO queue. The `PushReq` (push-request or write-enable) signal is asserted whenever the processor wishes to write data to the output queue. When the processor asserts `PushReq`, it also presents valid write data on the queue interface's data lines. If `PushReq` is asserted and `Full` is not, the processor assumes that the external FIFO queue has latched the data and that the write transaction is complete. If the `Full` signal is also asserted in a cycle when the processor asserts `PushReq`, no data transfer takes place. Each transfer cycle is independent of the previous cycle(s).

Note that the Xtensa processor will continue to assert its push-request signal when the external FIFO queue indicates that it is full. The external FIFO queue should be designed so that this state does not cause the external queue to overflow; the external queue must ignore the push request when the queue is full.

The Xtensa processor can commit and buffer two queue-write instructions made to a full queue without stalling. A third queue write to a full queue will stall the processor. This buffering ensures that instructions that write to a TIE queue can be issued in a fully pipelined manner. The buffered data is subsequently written to the external FIFO queue when the queue is no longer full.

If the Xtensa processor is interrupted during an output-queue access, all committed output-queue writes will still be performed without data loss.

The following TIE code illustrates the previous points. It instantiates an output queue called `QOUT`, and an instruction, `WRITEQ`, that writes to the queue.

```
queue QOUT 32 out
operation WRITEQ {in AR art} {out QOUT} [
     assign QOUT = art;
}
```

Figure 24–140 shows a basic output-queue write instruction that commits in cycle 5 and pushes data onto the external FIFO queue during cycle 5 by asserting the PushReq signal and presenting the queue data on the TIE_QOUT lines. Assertion of the PushReq signal indicates that the external queue must latch the data presented on the TIE_QOUT lines on the rising edge of the clock between cycle 5 and cycle 6.

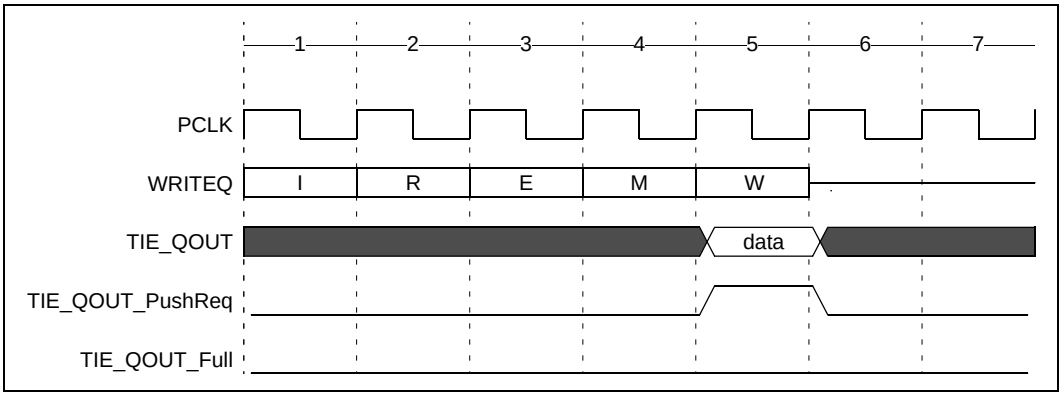


Figure 24–140. Output Queue Write in W Stage, Data Available in Following Stage

Output Queue Write to Full Queue

Figure 24–141 shows two queue-write instructions that commit in cycles 5 and 6, even though the output queue indicates that it is currently full (as indicated on TIE_QOUT_Full in cycle 5). The third queue-write instruction stalls during cycles 7 through 9 because the external queue is full and only two queue writes are buffered internally by the processor. The stall is released when the external queue releases its full signal, allowing the internally buffered queue writes to complete.

The first queue write appears at the output queue interface on TIE_QOUT in cycle 5, when the WRITEQ1 instruction commits. The second and third queue writes appear on TIE_QOUT in cycles 10 and 11 respectively, after the external queue negates its full signal.

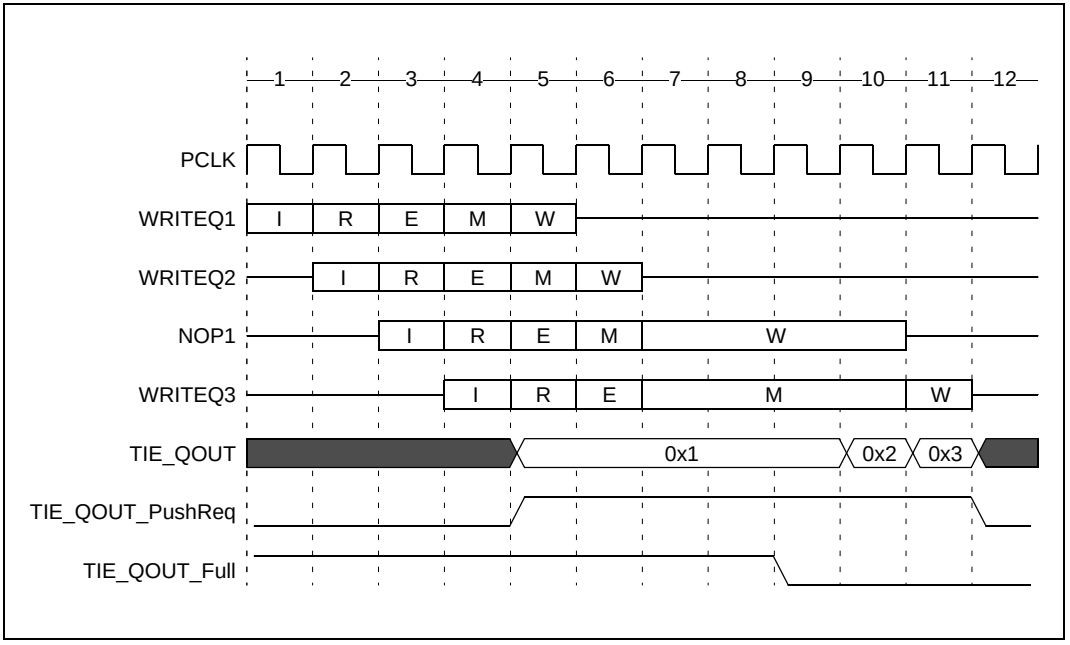


Figure 24–141. Output Queue Write Stalled

Note: If the output queue remains full and the stall lasts for sixteen cycles or more, there may be an extra stall cycle when the queue finally becomes non-full. This extra stall is required in some cases to guarantee that all memory enables can be turned off for power savings in the case of a long queue stall condition, and then turned back on correctly when operation resumes.

Output Queue Write and Interrupts

In Figure 24–142, an output-queue write commits in cycle 5 but the external FIFO queue currently indicates that it is full. In cycle 6, an interrupt causes the processor to kill the instructions in the pipeline but this action does not affect the already committed queue write. In cycle 8, the external queue indicates that it is no longer full, and the buffered queue write completes.

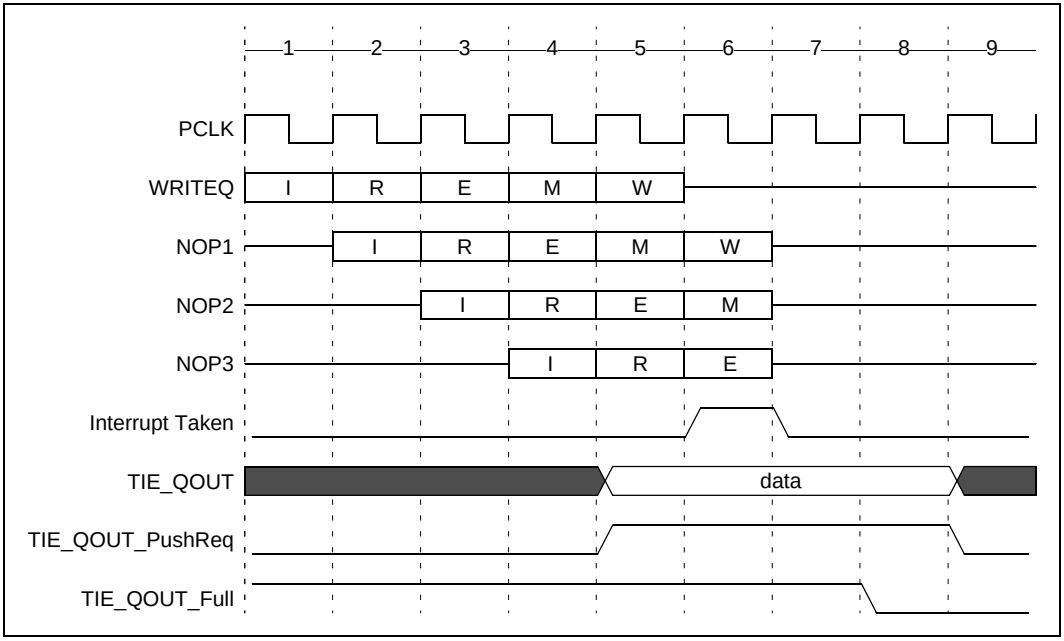


Figure 24–142. Interrupting Output Queue

TIE Queue Interface Recommendations

The TIE input- and output-queue interfaces have been designed to mate directly to synchronous FIFO devices. The TIE queue interfaces are synchronous in that all data transfers between the Xtensa processor and the external queues occur on the rising edge of the processor's internal clock signal. There should be no combinational paths from `PopReq` to `Empty` in input-queue interface logic and from `PushReq` to `Full` in output-queue interface logic. Thus, if an input queue becomes empty as a result of a pop request from the processor, the queue should assert its `Empty` signal in the next clock cycle, following the clock edge marking the data transfer. Similarly, if an output queue becomes full as a result of a push request, it should assert its `Full` signal in the next clock cycle after the clock edge, when it accepts the data from the processor.

The Xtensa processor can assert its pop-request signal when an input queue is empty, or it can assert its push-request signal when an output queue is full. External queue designs must protect the queue FIFOs against input-queue underflows and output-queue overflows by ignoring the processor's requests when an input queue is empty or when an output queue is full.

The Xtensa processor should be the only device connected to the read port of an input-queue FIFO or the write port of an output-queue FIFO. TIE queue interfaces are not designed to connect to external queues that are shared between multiple clients. During

normal queue operation, the status of an input queue must change from “not empty” to “empty” only in response to a pop request from the Xtensa processor. Similarly, the status of an output queue must change from “not full” to “full” only in response to a push request from the Xtensa processor. The only situation where it may be appropriate to violate this particular restriction (for both input and output queues) is during a system flush. Such a flush might occur, for example, before data processing begins on a new data stream.

System designers must ensure that the Xtensa TIE queue interfaces are used in a manner consistent with the recommendations outlined in this section. Not doing so will likely produce unexpected or incorrect queue behavior.

24.11.4 TIE Queues Compared to the XLMI Port

External queues can also be connected to Xtensa processors through an XLMI port. To choose between the XLMI port and TIE queue interfaces for external FIFO queue connections, consider the following points (in addition to other system constraints):

- FIFO queues connected to an XLMI port are accessed via memory-mapped load and store instructions, which can be either standard Xtensa instructions or designer-defined TIE instructions. External queues attached to the processor through TIE queues *must* employ designer-defined push and pop instructions because the TIE queue interfaces are implicitly addressed by TIE instructions and are not explicitly mapped into the processor’s address space. The designer should consider whether memory-mapped or TIE queue push and pop operations are the more appropriate usage model. The processor can transfer data to or from only one XLMI-attached queue during any given cycle (because there’s only one XLMI port per processor), but it can transfer data to and from several TIE queues during each clock cycle. XLMI-attached queues require no additional I/O lines other than the XLMI port but each additional TIE queue interface adds I/O lines to the processor. For some system designs, the interface choice won’t matter.
- Xtensa processor configurations cannot have more than one XLMI port, but they can have any number of TIE queues. If a FIFO queue is attached to the XLMI port, it must share that port with any other XLMI-attached devices, which can result in bandwidth issues. TIE queue interfaces are not shared. Therefore an external queue attached to a TIE queue interface always receives the full bandwidth of that interface.
- If the processor tries to read from an input FIFO queue when it’s empty, an XLMI-attached FIFO queue will stall the associated I/O operation. The stalled I/O read will immediately stall the processor’s pipeline as well and this stalled state is not interruptible. TIE queues can stall the processor’s pipeline but interrupts can still occur and the processor will service these interrupts when a TIE queue is stalled.

- If the processor executes a store to the XLMI port immediately followed by a load from an XLMI-attached FIFO queue, the store is buffered and the load occurs first. If this load causes an I/O stall because the addressed FIFO queue is empty, a resource deadlock will freeze the processor's pipeline. The XLMI load cannot complete because the FIFO queue is empty and the store to the FIFO queue cannot take place because the pipeline is stalled.
- All XLMI-attached devices including FIFO queues must handle speculative reads over the XLMI port. To do this, XLMI-attached devices must observe and use the XLMI control signals that indicate whether a read has committed or flushed. The Xtensa processor automatically handles speculative reads for TIE input-queues.
- The XLMI port width is restricted to be the same width as the configured local-memory interfaces (32-, 64-, 128-, 256-, or 512-bits wide). TIE queues can be 1 to 1024 bits wide and different queues in the same processor configuration can have different widths.

24.11.5 TIE Queue and Port Wizard

The Xplorer IDE provides a simple wizard for creating ports and queues without the need to write any TIE.

After specifying the width and direction of each port or queue, the wizard generates a TIE file that gets included in the processor configuration. A software project with test inputs is also created which can be run to exercise the created interfaces.

As the wizard output is a TIE file, this can be edited or deleted from the configuration and new interfaces can be added by re-running the wizard as many times as needed.

24.11.6 TIE Lookups

TIE lookups are ideally suited to perform table lookups when the lookup table must be implemented external to the processor, either because the table is large or because of the dynamic nature of the table data. Figure 24–143 shows the interfaces that are added to the Xtensa processor for each TIE lookup. It is useful to compare the lookup interface to a ROM interface, which has an address, read enable, and return data. The lookup has an output interface of designer-defined width (TIE_<name>_Out)—analagous to a ROM address—and a 1-bit request (TIE_<name>_Out_Req), which is analagous to a ROM's read enable. There is also an input interface (TIE_<name>_In) of designer-defined width that is analagous to a ROM read data port. In addition, there is an optional arbitration signal (TIE_<name>_Rdy), which is used to NACK (not acknowledge) the lookup request. If TIE_<name>_Out_Req and TIE_<name>_Rdy are both high at posedge clock, then TIE_<name>_In is sampled by Xtensa at the posedge clock <N> cycles later.

Note that you provide pipeline stages for the output and input of the lookup. N, the "latency" of the lookup, is the difference between the stage in which the output request is sent and the input is sampled.

The Xtensa processor will assert the output port Req and read the input port Rdy in the same cycle. When both the request and the ready signal are high in the same cycle, the request is considered to have been accepted by the external device. The input data port will be sampled N cycles later. Thus the Rdy signal is associated only with the request and not with the response. After the external device has accepted the request, it must provide a response after N cycles, where N is the latency of the lookup. If Req is high and Rdy is low, the external device is not accepting the request. In this case, the instruction using the lookup will typically stall and assert Req until Rdy is high. On the other hand, if the processor takes an exception or interrupt, it may de-assert the lookup request.

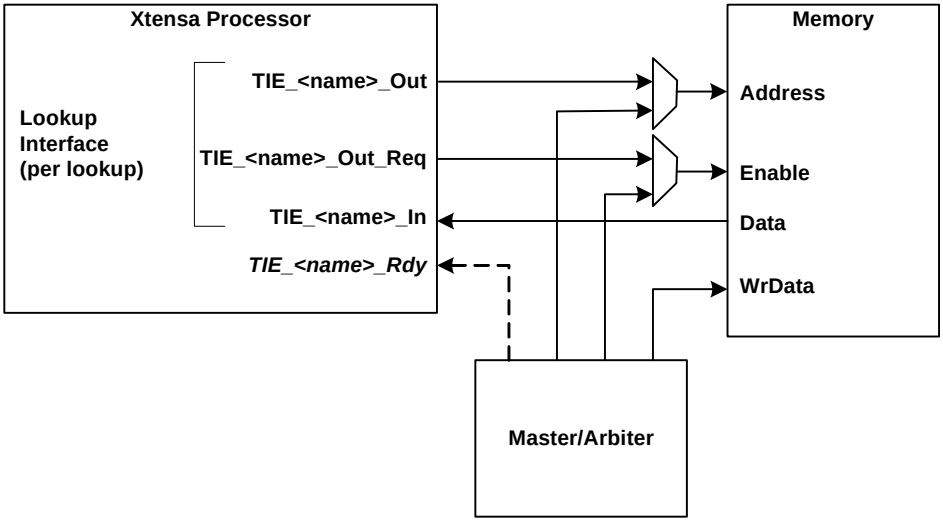


Figure 24–143. TIE Lookup Block Diagram

The ROM analogy is conceptually useful to achieve a basic understanding of TIE lookups, but there are a number of important differences as well as additional features to TIE lookups.

- TIE lookups are not extra memory interfaces. The output request can be considered an address with respect to the external hardware, but it is not treated as a load/store address by the processor hardware or by the software. For example, TIE lookups are not subject to the processor memory ordering semantics or symbolic linking.

- Similar to ROMs, TIE lookups have no predefined or built-in mechanism to initialize the memory values in the lookup table; there is no TIE construct to store values to a lookup table. Figure 24–143 shows an example system, where there is an external master and arbiter that is capable of writing the external memory. To facilitate this sharing of the lookup memory, an option arbitration signal (TIE_<name>_Rdy) can be configured. The arbiter can use this signal to block accesses from the processor until it has completed its initialization of the memory.

TIE lookups also offer greater flexibility than ROM interfaces, including the following:

- Lookups can occur during designer-defined pipeline stages for the output (e.g. address) and input (e.g. read data), whereas ROM interfaces have fixed stages for the address and data, and limited latency configurability.
- A designer-defined instruction can use multiple lookups, either in parallel or serially. For example, the return data of a lookup can be used to look up data in a second array.

TIE Lookup Example: Lookup Coefficients

In this example, a lookup interface is used to access a coefficient table that is external to the processor. The following snippet of TIE code describes the lookup interface and an instruction that multiplies a register value with a coefficient obtained through the TIE lookup. The table has an 8-bit index, corresponding to 256 entries. The return data is 256-bits wide and is valid in the M-stage. The lookup also uses the optional arbitration signal to allow an external agent to NACK the processor's request. The instruction, mul_coeff, multiples an input register value with the return data from the TIE lookup. The multiplication is performed in the W-stage, even though the table coefficient is available in the M-stage. (In this example the designer knows that the lookup timing delay will take too long to allow a multiplication to occur during the same cycle as the lookup.) Both the external delay of the TIE lookup, as well as the internal delay of the operation are defined by the designer.

Note: As a general rule, it is generally safest to register the lookup data before operating upon it, as illustrated in this example:

```
lookup TBL {8, Estage} {256, Mstage} rdy

operation mul_coeff {in AR index, inout AR acc}{out TBL_Out, in TBL_In}
{
     assign TBL_Out = index;     // Table index comes from AR regfile
     assign tmp = acc * TBL_In;  // Multiply reg with coef. from table
     assign acc = tmp[287:256];  // Keep 32-bits
}
schedule tbl {mul_coeff}
{
     use index Estage;
```

```
        use acc   Wstage; // Multiply is performed in the W-stage
        def acc   Wstage; // because the lookup data is late in M-stage
    }
```

Figure 24–144 shows an example timing for this TIE instruction and lookup. A table lookup request is issued starting in cycle 3, as indicated by the TIE_TBL_Out_Req signal assertion. The index (or address) is driven onto the TIE_TBL_Out interface. Note, the output delay of the TIE_TBL_Out interface is dependent on the designer-defined operation. In this case, there is little logic delay to generate this index, but other instructions can employ a more complicated expression for generating the index, which would incur a greater delay.

In cycle 3, the lookup arbitration indicates that the lookup is not ready (TIE_TBL_Rdy is low), and the lookup has failed. In cycle 4, the mul_coeff instruction stalls in the processor pipeline to wait for the lookup interface to become available. As shown in the transition from cycle 3 to cycle 4, the lookup interface outputs are not guaranteed to be held constant across cycles, or even re-issued. For example, an interrupt could cause the request to be aborted.

The lookup interface becomes available in cycle 5 as indicated by the assertion of TIE_TBL_Rdy. Because the lookup was defined with a single-cycle of latency, the lookup data is available in cycle 6 where the processor accepts it.

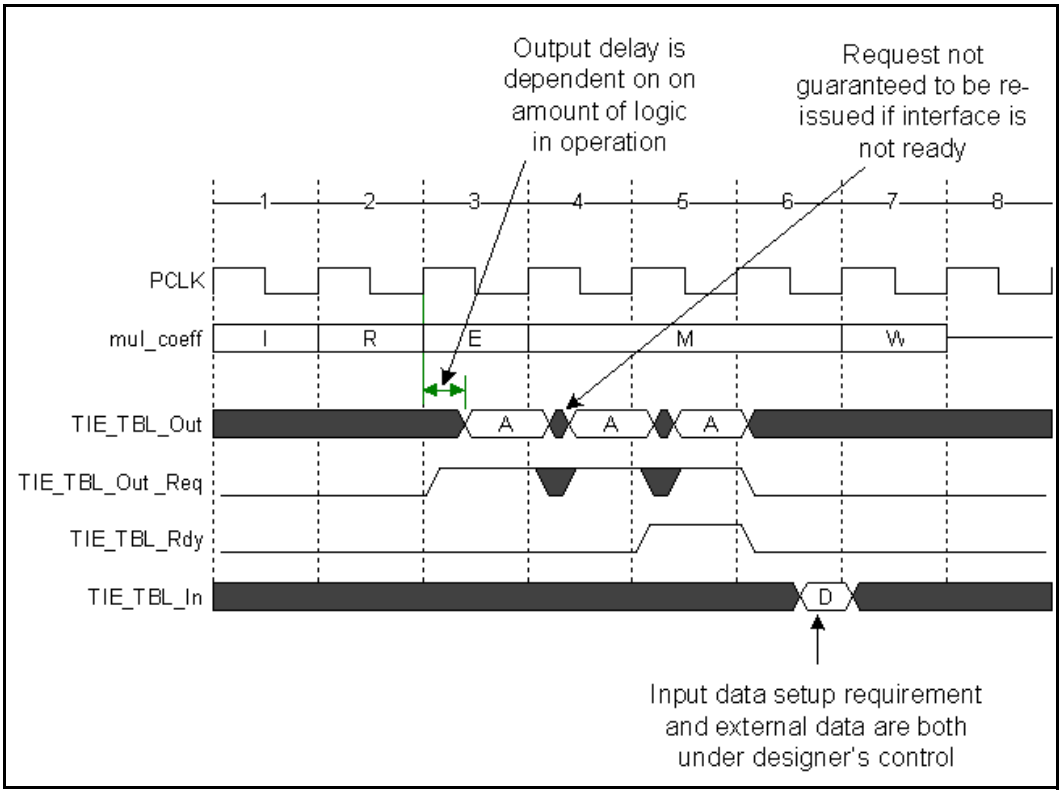


Figure 24–144. Lookup Example

Note: For multiple operations that use the same TIE lookup, grouping into a TIE semantic block may be beneficial to optimize the lookup timing and to share any common logic.

24.11.7 TIE Lookup Arbitration

The optional Rdy input allows an external lookup to be shared between several Xtensa processors or other blocks using some arbitration. Cadence recommends that the status of the Rdy signal be pre-determined by the arbitration logic, before the Xtensa processor asserts the Req signal. This will prevent a direct combinational path from Req to Rdy, which is undesirable for timing and physical design considerations. Rdy can also be used when the external device is not pipelined, (that is, it cannot accept a request every cycle). In this case, the device can de-assert Rdy after every request until it is ready to accept the next one.

The high level modeling of the TIE lookup memories in the ISS is targeted towards fast simulation speed. As a result, simulating TIE lookup memories in the ISS does not support arbitration. This means that the arbitration signal (TIE_<name>_Rdy) is not supported in standalone ISS modeling. Also all writes to TIE lookup memories in the ISS will be committed in the W stage.

Use XTMP, XTSC, or the lower level RTL level simulation environment to model arbitration for TIE lookups using the arbitration signal (TIE_<name>_Rdy).

24.11.8 TIE Memories

TIE lookup memories are supported with read and write functionality, making them ideally suited to perform localized scratch pad memory operations. Figure 24–145 shows the interfaces that are added to the Xtensa processor for each TIE lookup. It is useful to compare the lookup interface to an SRAM interface, which has an address, enable, write enable, write data and read data. The TIE Memory has an output interface of designer-defined width (TIE_<name>_Out), which is analogous to a combination of SRAM write data, SRAM address and a 1-bit request. There is a request signal generated (TIE_<name>_Out_Req), which is analogous to a SRAM's enable signal. There is also an input interface (TIE_<name>_In) of designer-defined width that is analogous to a SRAM read data port.

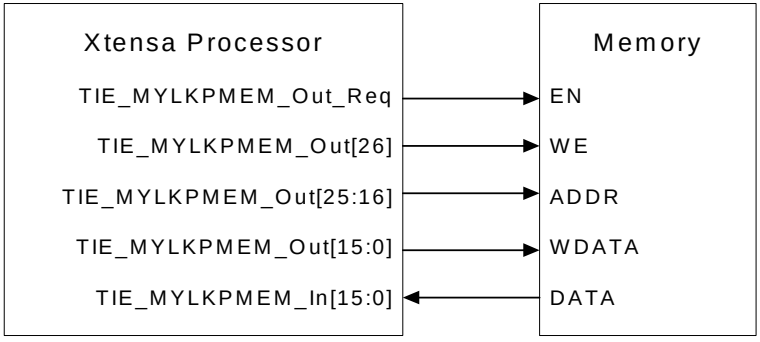


Figure 24–145. TIE Lookup Memory Connectivity Diagram

The TIE lookup memory feature is fully supported by ISS support, hence allowing simulation within XTSC. To function as a RAM, `xtsc_lookup` has to made aware of the location within the lookup address of the RAM address, write data and write strobe bit fields. Definition needs to be made if the write strobe is active high or low (this information is specified by the TIE code). `xtsc_lookup` get this information from the `write_data_lsb`, `write_strobe_bit`, and `active_high_strobe` parameters. If these parameters are not explicitly set, their default values will result in `write_data_lsb`. Assuming the write data bit field occupies the least-significant bits of the lookup address, the write strobe bit is active high and occupies the most significant bit, and the

RAM address occupies the bits in between. Because `lookup_ram` has exactly these settings in the TIE code, there is no need to explicitly set these parameter values in `xtsc_lookup_parms`.

If TIE code for your lookup RAM does not use these default values, you need to change the settings in the `xtsc_lookup_parms` object before calling the `xtsc_lookup` constructor. For example:

```
lookup_parms.set("active_high_strobe", false);
xtsc_lookup lram("lram", lookup_parms);
```

The following code shows an example definition of TIE Memory objects (matching naming used in Figure 24–145):

```
property lookup_memory MYLKPMEM

lookup MYLKPMEM {27, Wstage}{16, Wstage+1} // 1024 locations, 16-bit
// ( 1-bit write enable, 10-bit Addr, 16-bit Write Data)

operation MYLKPLD {in AR a, out AR d}{out MYLKPMEM_Out, in MYLKPMEM_In}
{
     assign MYLKPMEM_Out = {1'b0, a[9:0], 16'b0};
     assign d = MYLKPMEM_In;
}
operation MYLKPST {in AR a, in AR d}{out MYLKPMEM_Out, in MYLKPMEM_In}
{
     assign MYLKPMEM_Out = {1'b1, a[9:0], d[15:0]};
}
```

The following example program does many writes and reads to the RAM and checks to make sure the value read back matches what was written. It keeps a count of any errors and prints the total at the end. Following is the main() function from the program. Notice the calls to MYLKPLD() and MYLKPST():

```
#define ADDR_BITS 10

int main(int argc, char *argv[]) {
     int offset;
     int errors = 0;

     LOG("Xtensa core - starting.

     // Write all
     for (offset = 0; offset<(1<<ADDR_BITS); offset++) {
          MYLKPST(offset, offset);
     }
```

```
        // Read all
        for (offset = 0; offset<(1<<ADDR_BITS); offset++) {
             errors += MYLKPLD(offset) != offset;
        }

        // Write, read interleaved
        for (offset = 0; offset<(1<<ADDR_BITS); offset++) {
             MYLKPST(offset,offset);
             errors += MYLKPST(offset) != offset;
        }

        LOG1("Total errors = %d

        return errors;
    }
```

In this example, the standard output of core0 is sent to core0_output.log (shown here): 36850: Xtensa core - starting. 90303: Total errors = 0

Note: TIE_MYLKPMEM, which is an example naming. TIE_MYLKPMEM_Out_Req is a port automatically generated within the TIE Compiler supporting TIE lookup memories.

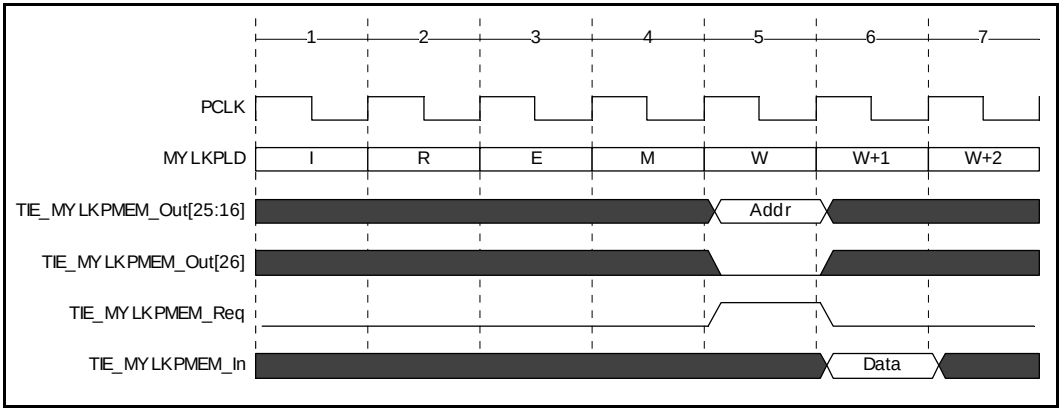


Figure 24–146. Lookup Memory Read Access Timing Diagram

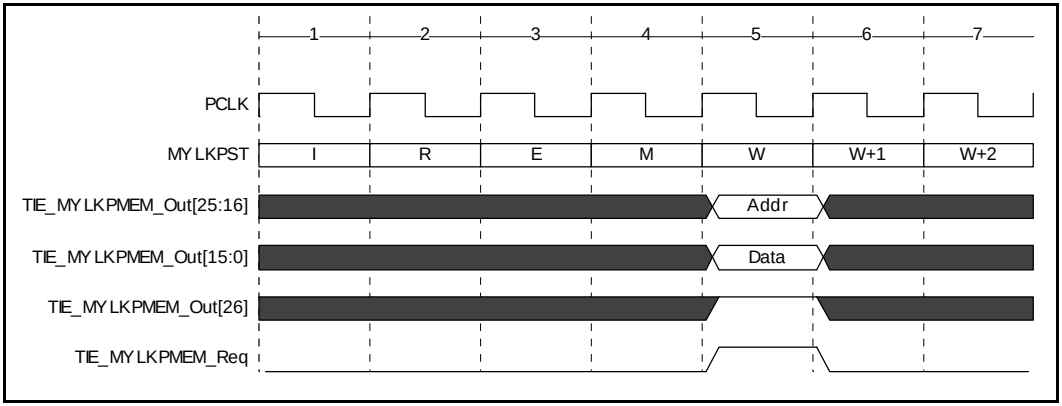


Figure 24–147. Lookup Memory Write Access Timing Diagram

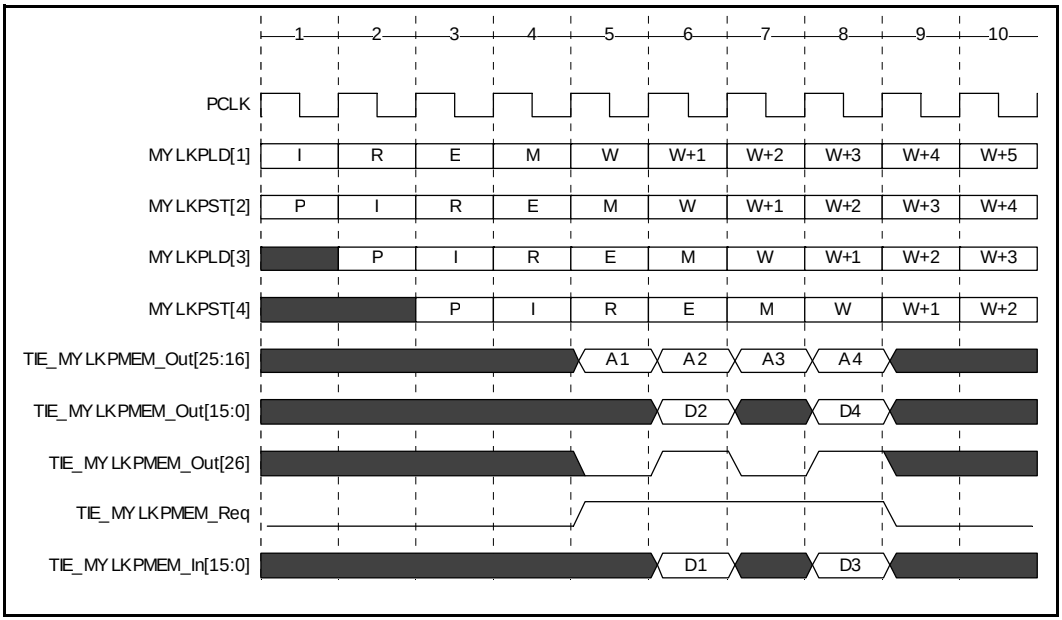


Figure 24–148. Lookup Memory with Back-to-Back Read, Write, Read, Write Access

25. Xtensa EDA Development Environment and Hardware Implementation

This chapter describes the EDA development environment to perform synthesis, place-and-route, gate-level simulation, power analysis, and formal verification. For complete details about these topics, refer to the *Xtensa Hardware User's Guide*.

Notes on physical implementation flows:

- Designers are free to use their preferred EDA tools and flows to implement Xtensa processors. Cadence does not prohibit usage of any particular tool or vendor. Cadence provides scripts and reference flows for most leading EDA tools as a convenience to the designer. The designer may use any part or all of the EDA scripts provided.
- Since Xtensa processors are configurable, each processor potentially is different and the attributes of each processor may vary widely. Therefore the area, power, or frequency of any given Xtensa processor cannot be guaranteed by Cadence and will be unique to the designer and dependent upon the designer's choices of implementation flow and process technology and memories.

25.1 Design Environment

Cadence provides a set of EDA scripts for Xtensa processors to perform logic synthesis, place and route, extraction, static timing analysis, gate-level simulation, power analysis, and formal verification.

The starting point for exercising the implementation flow is to modify a CAD specification file that captures design targets, library related information, and tool-specific parameters. The design targets include target clock period, clock skew, clock jitter, clock transition time, and clock insertion delay. The library related information specifies certain library elements (cells and pins) and the location of site-specific installation of library views. The tool-specific parameters guide the tool scripts provided by Cadence.

After customizing the CAD specification file, the designer can synthesize the processor RTL, perform physical design and timing verification, and simulate the post-layout gate-level netlist and perform power analysis. These automated steps give the designer a quick and accurate view of the timing, power, and area achievable in the designer's tool environment.

Figure 25–149 illustrates the Xtensa processor implementation flow. As shown in the figure, the starting point of the flow is the Cadence provided RTL. The front-end synthesis takes this as its inputs to produce a gate-level netlist. The back-end tools take in the

gate-level netlist to produce an optimized placed and routed design. The parasitics are extracted by extraction tool and are used with the placed and routed design to generate a timing report using static timing analysis tool. Formal verification, functional verification and power analysis can be performed after each of these major steps, i.e. post-synthesis as well as post-route.

Cadence supports implementation tools from all the major EDA vendors. The front-end synthesis scripts are provided for Cadence RTL Compiler (RC) and Synopsys Design Compiler (DC). The back-end tools scripts are provided for the Cadence Encounter Digital Implementation (EDI) tool. Extraction scripts are provided for Cadence QRC. Finally, Cadence Tempus and Synopsys PrimeTime-SI scripts are provided for post-layout timing verification.

To automate gate-level simulation, Cadence provides scripts for Synopsys VCS, Cadence IUS, and Mentor Graphics Verilog MTI. The same diagnostic used for gate-level simulation can optionally generate an activity (toggle) file. Automated scripts for both Synopsys Power Compiler and Cadence RTL Compiler are provided to read in this toggle information and analyze the power of a gate-level netlist.

Cadence provides scripts to perform RTL-to-gates and gates-to-gates formal verification. The scripts are provided for the Cadence Conformal-Ultra formal verification tool. The flow to formally verify RTL to post-layout gate must be exercised in two steps: RTL-to-gates (post-synthesis gates) and gates-to-gates (post-synthesis gates to post-layout gates).

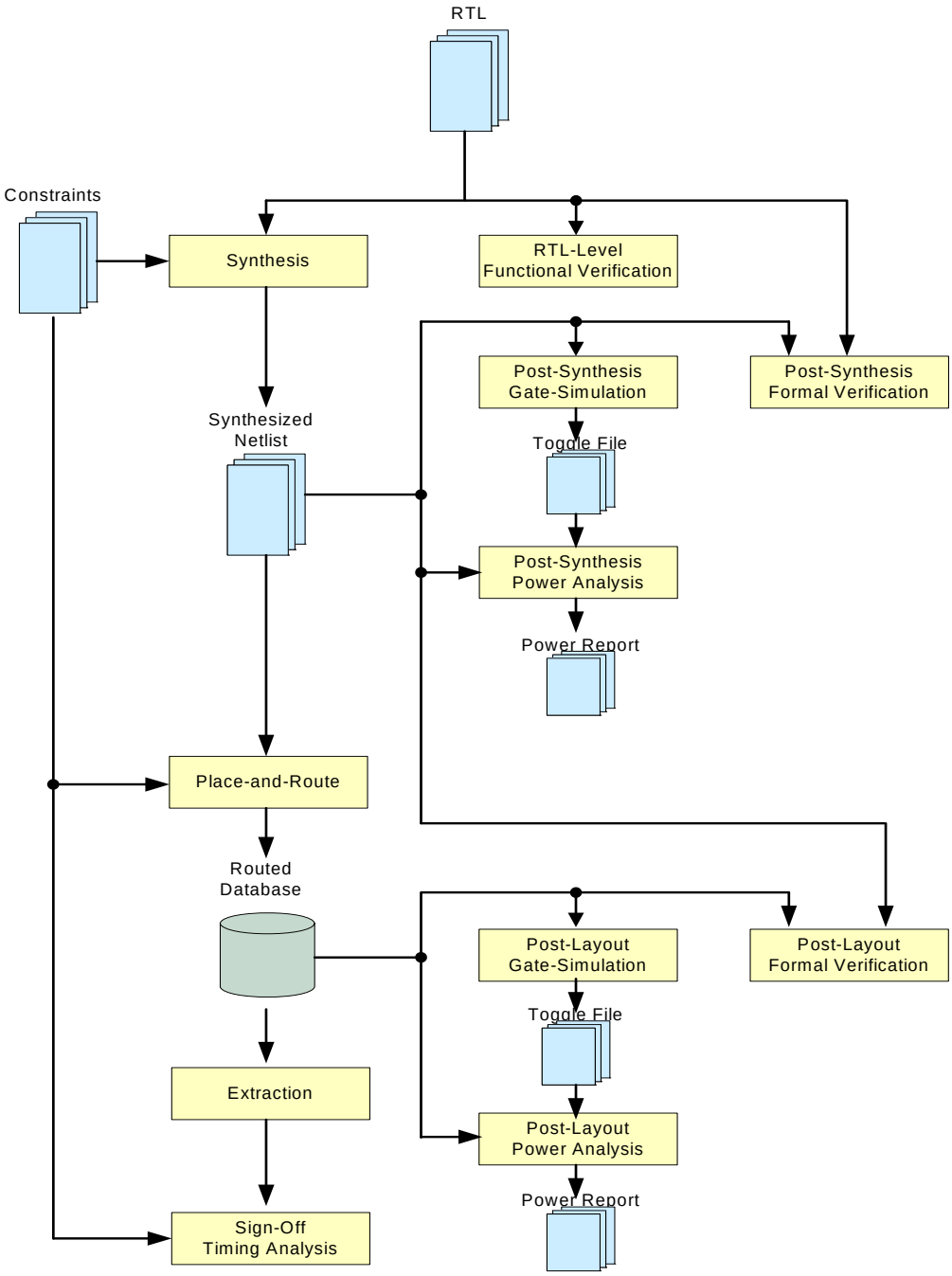


Figure 25–149. Xtensa Processor Implementation and Verification Flows

25.2 Front-End Synthesis

The processor RTL is synthesized using Cadence RTL Compiler or Synopsys Design Compiler to generate a timing-, area-, or power-optimized gate-level netlist. Cadence provides a set of Design Compiler and RTL Compiler reference scripts that are optimized for the Xtensa processor core. The associated scripting infrastructure allows designers to specify any standard-cell library, design constraints, and optimization switches.

Cadence scripts automatically generate the default I/O constraints for the Xtensa processor. The designer can further modify these constraints. The timing defaults to PIF ports or Bus Bridge ports based on AC timing specifications. Memory-port timing defaults are derived using industry-standard memory compilers. For optimum results, the designer should adjust the provided default timing budgets to reflect specific memory and system requirements.

If memory hard macro library views are available, the designer can synthesize the entire processor plus memory subsystem, which will give more accurate results than editing the memory I/O budget file directly. Simulation and formal verification of the resulting netlist is also supported. However, Cadence does not provide automated scripts to perform placement and routing of the memory subsystem.

The front-end synthesis scripts provide flexible compilation strategies using physical information from the libraries – a top-down multipass approach using either DC Topographical, or RC PLE. The synthesis scripts can use Cadence-provided design primitives or Direct primitives that target the Cadence ChipWare library or Synopsys DesignWare. The optimization switches in the CAD specification file allow developers to perform speed, power, area, and turnaround-time trade-offs. Cadence synthesis scripts automatically identify and implement register rebalancing (retiming) for TIE semantic modules (designer-defined processor execution units) that require multiple pipeline stages.

The scripts use a path-grouping technique to ensure that flop-to-flop paths are optimized independently of I/O paths within the processor RTL. This technique optimizes internal paths in the processor, even in presence of approximate timing budgets that might evolve during the design process to suit changing system requirements.

The Xtensa processor design aggressively gates the clock to minimize dynamic operating power. During synthesis, the scripts model buffering delays from the output of a gated clock to the registers by specifying maximum delay constraints to the enable pins of the gated clock.

The scripts also optionally support design-for-testability (DFT) scan insertion and chain stitching. The maximum length of any single scan chain can be controlled.To increase test coverage if synthesizing with memory hard macros, automatic test point insertion on memory I/O paths is provided.

25.3 Back-End Layout

The Cadence Xtensa Processor Generator (XPG) creates scripts for the Cadence Encounter Digital Implementation layout tool. The scripts include optimization switches to manage timing, power, congestion, and cell area. The layout scripts also perform scan-chain deletion and restitching to avoid scan-path wiring jumbles resulting from placement. Scripts are provided to complete clock-tree synthesis, timing optimization, and routing. The XPG also creates a script for a Cadence-proprietary tool (the XT floorplan initializer) that creates the initial floorplan.

25.4 Parasitic Extraction

Parasitic extraction takes a routed design database, performs 3D RC extraction, and produces parasitic data in a SPEF file format. The supported vendor tool is Cadence QRC.

25.5 Timing SI Verification

The timing-verification step takes a routed netlist, timing constraints, and the extracted SPEF file and then performs on-chip variation (OCV) or min-max timing analysis and generates delay and timing report files. The supported tools are Cadence Tempus and Synopsys PrimeTime-SI.

25.6 Multi-Mode Multi-Corner (MMMC) Support

To better handle the variability of processes at 28nm and below, MMMC support is provided for EDI layout, QRC extraction, and Tempus and PrimeTime timing analysis. The designer edits a single file to specify the different scenarios (process corner and wiring capacitance combinations) to be used at each step of the flow, and the scripts automatically take account of this information.

25.7 Power Characterization and Optimization

Cadence provides scripts to perform gate-level power analysis on Xtensa processor cores. The same diagnostics used to perform functional verification also generate toggle activity for the core. This toggle file, in conjunction with a gate-level netlist and loading information, is used to analyze a processor design's total power consumption. These power numbers are reported in terms of gate, net, and leakage components. Other provided scripts perform power optimization on a netlist, which is achieved primarily by downsizing gates along non-critical paths. Both power analysis and optimization are performed with either Design Compiler or RTL Compiler. Design Compiler requires a Power Compiler license.

The low power flow option selectable from the CAD specification file may insert additional clock gates, optimize the clock tree, and add gates to isolate the datapath operands to reduce power during front-end synthesis. The same optimization switch directs the back-end layout tool to perform power-aware placement, optimization, and routing.

If the designer wants to skip gate-level simulation while still performing gate-level power analysis on their program, a hybrid approach is provided: toggle activity is taken from an RTL-level simulation and applied to a gate-level netlist power analysis run. This provides faster turn-around time with a minimal decrease in accuracy compared to a full gate-level simulation.

Cadence Voltus scripts are provided for IR drop and power analysis.

25.8 Hard Macros

Using hard macros for certain components within the Xtensa processor can improve timing performance, reduce the die size, or alleviate wiring congestion. The scripting infrastructure allows developers to easily use hard macros for the processor's register file, write buffer, and integrated clock-gating cell. A file containing placeholder modules is provided to allow easy insertion of any of the hard macros listed above. The designer can also replace any register file with a hard macro. The supplied scripting infrastructure automatically incorporates hard macros in the processor design hierarchy during synthesis. The gate-level netlist with any clock-gating hard macros can then be taken through the rest of the Cadence-provided flow. The gate-level netlist with register file macros require manual intervention to create the floorplan, place the macros, and complete placement and routing of standard cells.

25.9 Diagnostics and Verification

Several diagnostic and verification tools are included in the Xtensa processor package.

25.9.1 Xtensa Processor Core Verification

Cadence has executed trillions of cycles of architecture-verification diagnostic tests to verify the execution of instructions specified in the Xtensa processor's Instruction Set Architecture (ISA), to verify the processor's micro-architecture (testing for events that break the general pipeline flow such as exceptions, interrupts, or cache misses and to test for specific configuration options that operate independently of the processor's data path), and to trigger complex interactions across different parts of the processor core using random diagnostic tests. Cadence continuously runs co-simulation regression tests to correlate RTL and ISS (instruction-set simulator) simulation results. These exhaustive tests have produced a configurable, extensible processor core that is quite robust, as proven by hundreds of successful tapeouts.

25.9.2 Designer-Defined TIE Verification

Correct-by-construction verification of designer-defined TIE implementations is performed by writing test/verification diagnostics in C or C++ to exercise the new TIE constructs. The XPG automatically creates micro-architecture diagnostics to verify the processor-core-to-TIE interface. These steps ensure that designer-defined TIE instructions are implemented in the generated RTL *exactly as the designer described them*.

The Xtensa verification environment also supports formal verification (using Cadence Conformal-Ultra) to ensure that the designer-defined TIE references and semantics are logically equivalent.

25.9.3 Formal Verification

Cadence supports the Cadence Conformal-Ultra formal verification tool to compare adjacent points in the implementation flow. The adjacent points refer to RTL and post-synthesis gates or post-synthesis gates to post-layout gates. The formal verification scripts are provided to automate the formal verification flow for the Xtensa processor. Every attempt is made to ensure that formal verification scripts capture the best practices recommended. The scripts are designed to work for the majority of possible processor cores. However, additional commands may be necessary to get a particular processor configuration to pass formal verification. Also, the use of advanced optimization features such as retiming, clock-gate and data-gate insertion, clock tree optimization, FSM optimization may cause the formal verification tool to fail. Contact your formal verification tool provider for assistance with these hard to verify cases.

25.9.4 SOC Pre-Silicon Verification

A system-on-a-chip (SOC) design may consist of various intellectual property (IP) blocks integrated with one or several Xtensa processors. Pre-silicon SOC verification at the top level is accomplished using various methods. These methods include the Xtensa processor's Instruction Set Simulator (ISS) used with the Xtensa Modeling Protocol (XTMP) API environment, XTSC (the Xtensa SystemC package), RTL simulation, FPGA-based emulation, and hardware/software co-verification using third party tools. It is left up to the SOC design team to individually verify that IP blocks, other than the Xtensa processor(s) function as specified.

Cadence also provides System Verilog Assertion monitors to check for violations of interface protocol during simulation. Verifying SOC connectivity to the Xtensa processor(s) is an important phase in pre-silicon SOC verification. These monitors aid the verification and debugging of logic that interfaces to the Xtensa processor(s), either through the PIF or through the local memory interfaces or the XLMI port. The monitors also pro-

vide immediate verification feedback if the system violates the interface specifications. Consequently, the verification engineer need not write diagnostics for the Xtensa processor's interfaces and can instead spend more time on system-level testing.

25.9.5 SOC Post-Silicon Verification

The Xtensa processor is a fully synchronous design that is completely compatible with full-scan and partial-scan manufacturing test methodologies. The scan-insertion and ATPG flow is fully compatible with all of the industry-standard tools currently in use.

The Xtensa processor implements special instructions that directly read and write the cache data and tag RAM arrays. These instructions can be used for manufacturing tests on the embedded cache memories and for power-on self-test (POST).

For more detailed information about these topics, refer to the *Xtensa Hardware User's Guide*.

26. Bus Bridges

Xtensa processors perform traditional main memory and system accesses on a Processor Interface (PIF) port. The PIF is a configurable, high-performance, multi-master interface definition that is easily bridged to third party buses and interconnects. Cadence provides bus bridges to AHB-Lite and AXI.

Configuring the AHB-Lite or AXI bridge makes integration of Xtensa processor into an AMBA-based SOC easier. The processor and bridge are instantiated under the Xttop wrapper, thus they can be synthesized together and integrated as a single block in the system.

Note that the AHB-Lite bridge must be used in a multi-layer system configuration to support multiple masters.

26.1 Bus Bridges Features

When the AHB or AXI bus bridge is configured:

- It acts as a bus master to service the processor's outbound requests.
- It act as a bus slave if the processor is configured with the inbound PIF option, which gives other bus masters access to the processor's local memories.
- The bus bridge is configured to match the processor's configuration options, including:
  - Endianess
  - Implementation options such as scan options
  - Clock-gating options
  - Synchronous or asynchronous reset
  - Critical Word First option

26.2 Clocking

Three clocking modes are supported for both the AHB and AXI bus bridges as follows:

- The processor and bus clocks are synchronous and the same frequency.
- The processor and bus clocks are synchronous, but the processor clock frequency is an integer multiple faster than the bus frequency. Multiples up to a 4-to-1 clock ratio are allowed.

- The processor and bus clocks are asynchronous. It is possible to run the bus at a faster or slower speed than the processor. However the faster of the two clocks must not have a frequency greater than16 times that of the slower clock. The PIF asynchronous FIFO synchronizes PIF requests and responses.

The two synchronous modes are the highest performance modes of operation. An asynchronous operation incurs several cycles of latency, as described in Section 26.2.2, but offers the greatest flexibility in separately determining the optimal frequency for the processor and bus.

Section 26.2.1 explains the synchronous clock ratio mode of operation and Section 26.2.2 explains the asynchronous mode of operation.

Table 26–101 describes how the different clocking modes are configured and controlled.

Table 26–101. Clocking Summary

Mode	Asynchronous AMBA Interface Option	Strobe Input
Synchronous and same frequency	Not selected	Tied to 1'b1
Synchronous clock ratio	Not selected	Asserted to indicate phase of the bus clock
Asynchronous	Selected	Strobe input is removed

26.2.1 Synchronous Clock Ratios

In the synchronous clock ratio mode, the processor and bus clocks are synchronous, but the processor clock frequency is at an integer multiple of the bus clock. Ratios up to 4-to-1 are supported.

The design has a single clock input, which is connected to the higher frequency processor clock, and a strobe input that is used to select the rising edge of the bus clock. The rising edge of the bus clock must be synchronous with the processor clock rising edge. All timing specifications of the bridge are made with respect to the faster processor clock.

Note: No special synchronizing circuits are used to synchronize bus signals, as the two clock domains are considered synchronous.

Figure 26–150 shows the relationship between the Processor clock (CLK), bus clock (HCLK for AHB-Lite, ACLK for AXI), and strobe signal. Frequency ratios of 1:1, 2:1, 3:1, and 4:1 are supported.

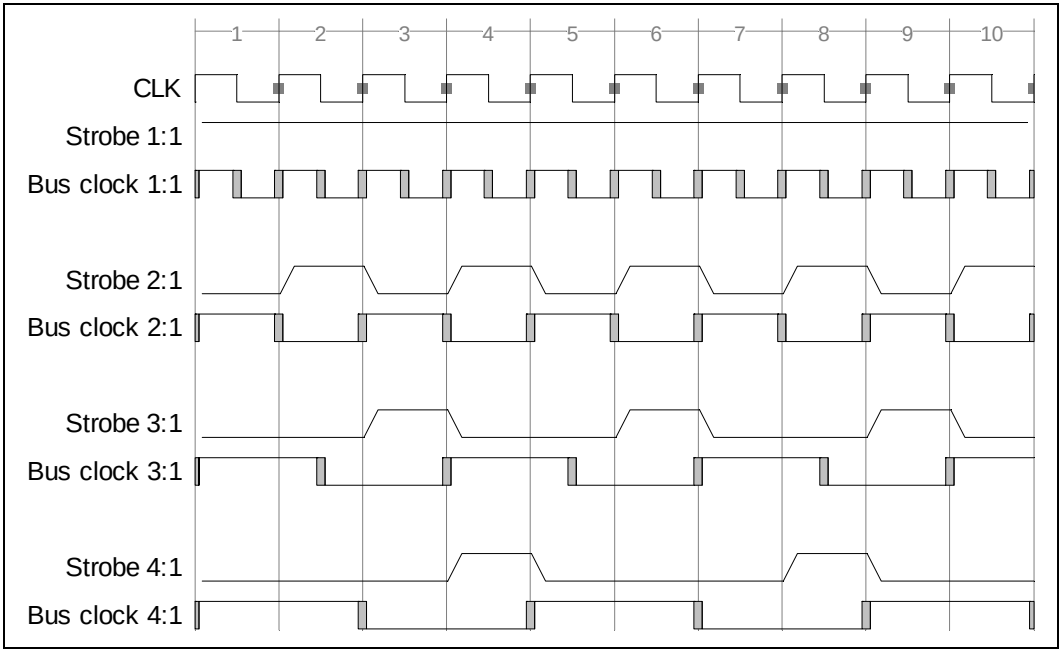


Figure 26–150. Clock Ratios

26.2.2 Asynchronous Clock Domains

A standard asynchronous FIFO is used to synchronize transfers between the processor and bus clock domains. The design is based on a paper presented at SNUG 2002 (Synopsys Users Group Conference, San Jose, CA 2002), titled "Simulation and Synthesis Techniques for Asynchronous FIFO Design", by Clifford E. Cummings.

A request and response FIFO pair is used for each of the master and slave ports of the processor, as shown in Figure 26–151.

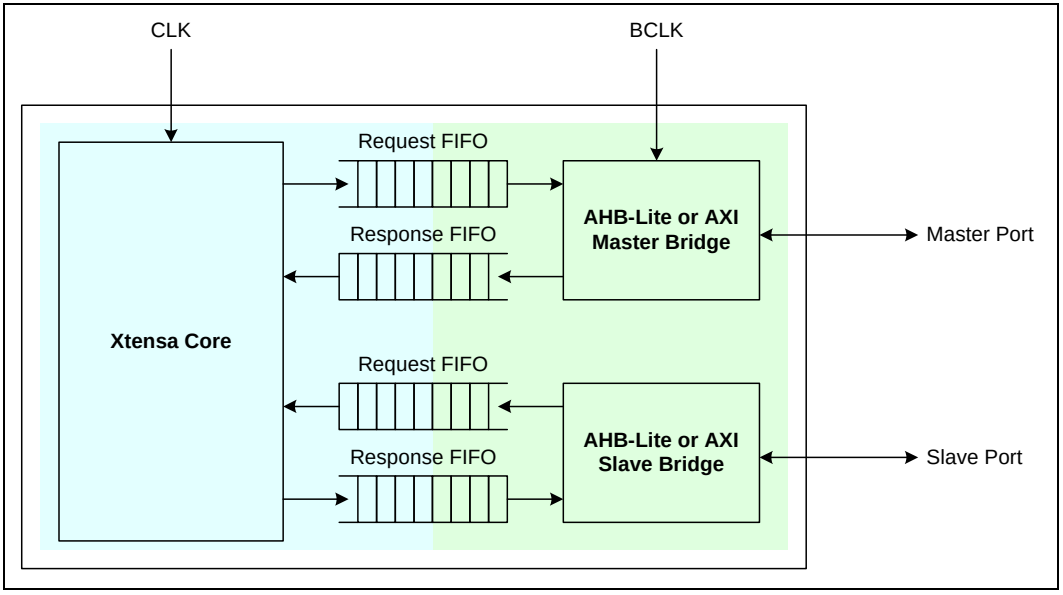


Figure 26–151. Xttop Block Diagram with Asynchronous AMBA Bridge Interface Option

The BCLK port shown in Figure 26–151 is connected to the bus clock as follows:

- HCLK for AHB-Lite
- ACLK for AXI

The asynchronous FIFO has the following properties:

- The FIFO has two clock domains:
  - The processor clock is called *CLK*.
  - The bus clock is called *BCLK*.
- The latency of each FIFO is one source clock domain cycle, and approximately 2.5 destination clock domain cycles. Read requests pass through a request and response FIFO, for a total additional latency of 3.5 processor clock cycles and 3.5 bus clock cycles. This latency is added to any additional latency through the AHB-Lite or AXI bus bridges.
- Each instance of the asynchronous FIFO has storage for eight entries.
- The storage is implemented as a register file, similar to the basic register files of the processor.

The asynchronous FIFO has the following requirements:

- The bus clock is allowed to be slower or faster than the processor clock, however the faster of the two clocks must not have a frequency greater than 16 times the frequency of the slower clock.

- CLK and BCLK must be truly asynchronous for correct operation.
  - The MTBF calculations of a synchronizer generally depend on an asynchronous relationship between the two clocks.
  - For example, the worst possible scenario would be a fixed integer ratio between the processor and bus clocks, with a phase relationship such that timing was violated every clock cycle for the synchronizing registers. This would lead to a higher probability of metastability and shorter MTBF.

26.2.3 Asynchronous FIFO Latency Components

The latency of a write into the asynchronous FIFO is comprised of the following:

- One cycle in the write clock domain to increment the write pointer.
- The write pointer is captured into the read clock domain. Some fraction of the read clock period is incurred as latency, depending on the phase relationship of the two clocks for each synchronization. This latency is reported as an average of 0.5 cycles in this document.
- A read-clock domain cycle to allow the synchronized write pointer to resolve any potential metastability issues.
- An additional read-clock domain cycle to compute the empty condition.

The latency of a read of the asynchronous FIFO is similar to the write latency, but is not included when considering the request or response latency of the asynchronous FIFO. The read latency has an effect on the utilization of the FIFO and the minimum number of entries required for unimpeded throughput. It is comprised of the following:

- One cycle in the read clock domain to increment the read pointer.
- The read pointer is captured into the write clock domain. Some fraction of the write clock period is incurred as latency, depending on the phase relationship of the two clocks for each synchronization.
- A write clock domain cycle to allow the synchronized read pointer to resolve any potential metastability issues.
- An additional write clock domain cycle to compute the full condition.

27. AHB-Lite Bus Bridge

The PIF-to-AHB-Lite bridge allows designers to integrate Xtensa processors into AMBA® 2.0 AHB-Lite systems. AMBA (Advanced Microcontroller Bus Architecture[1]) is an open-standard specification of a set of on-chip buses and interconnects. AHB (Advanced High-performance Bus) is one of the protocols defined within the AMBA specification, and AHB-Lite defines a subset of the AHB protocol that is targeted for single-master subsystems. Multi-master systems can be composed of multiple single-master subsystems, referred to as *layers* in the Multi-Layer AHB Overview (ARM® DVI 0045A, Section), as shown in Figure 27–152. Multi-layered systems are preferred over past generation shared bus implementations, due to the higher performance and clock frequency that can be achieved with the multi-layered approach.

For legacy systems with a shared bus implementation requiring the full-AHB protocol, the AHB-Lite specification indicates that “AHB or AHB-Lite can be used interchangeably” with the use of a “standard AHB master/slave wrapper” (ARM® DVI 0044A, p 4).

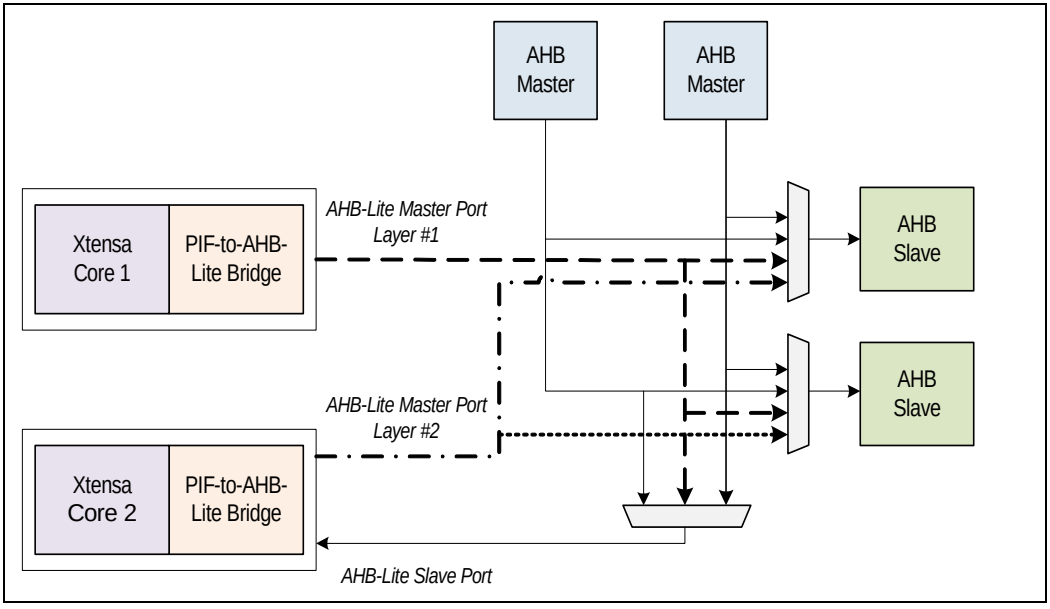


Figure 27–152. Example Multiprocessor System Diagram

1. AMBA® and ARM® are registered trademarks of ARM Limited

27.1 Configuration Options

The AHB-Lite bus bridge supports the following configuration options:

1. Processor Data Bus Width
   - 32-, 64- and 128-bit data bus widths are supported
2. Endianness
   - The bridge supports little and big endian configurations
3. AHB-Lite Slave
   - The AHB-Lite slave port is optional. The AHB-Lite slave port is used to access the processor's local memories directly from an external AHB-Lite master, such as a DMA controller. The Xtensa processors are capable of executing from an instruction or accessing data memory without stalling while a different instruction memory or data memory is being accessed through the bridge's AHB-Lite slave port.
4. PIF Request Attributes
   - The PIF interface of the processor can be configured to use PIF request attributes. It adds an additional request attribute output for a PIF master (POReqAttribute[11:0]), and input for a PIF slave (PIReqAttribute[11:0]). When the PIF Request Attributes option is selected, the PIF-to-AHB-Lite bridge adds an extra port, HXTUSER[3:0] to the AHB master interface. The bottom four bits of the POReqAttribute[11:0] signal are assigned to HXTUSER[3:0]. The PIF-to-AHB-Lite bridge also uses bits 4, 5, 9, and 10 of the POReqAttribute[11:0] signal to drive the HPROT[3:0] signal on the AHB bus. Refer to Section 27.4.3 "AHB Protection" for details. The PIF Request Attribute option does not make any changes to the AHB slave interface.
5. Critical Word First
   - When the processor is configured with the Critical Word First option, it may send out WRAP requests on the AHB-Lite bus with the starting address not aligned to the burst size. The subsequent addresses will increment in a simple wrapping fashion.
   - If the Critical Word First option is not selected, all AHB burst requests will be issued on the AHB-Lite bus with start addresses aligned to the burst size.
6. Clocking mode
   - Synchronous and asynchronous modes are available.

The following third party documents describe the AHB-Lite bus protocol. These documents are available from ARM®.

- "AMBA™ Specification" (rev 2.0),ARM Ltd,; Document Reference Number: IHI 0011A, Issue A
- "AHB-Lite Overview",ARM Ltd,; Document Reference Number: ARM DVI 0044A

27.2 AHB-Lite Specification Compliance

The bridge implements the subset of the AHB protocol defined as AHB-Lite. The bridge deviates from the defined specification by not allowing early terminated requests on the optional AHB-Lite slave port.

The AHB master bridge never initiates an early burst termination. It will not terminate a burst operation for any reason, including the reception of read errors from the slave.

27.3 Reset

The AHB-Lite bus bridge must be reset with the same active-high reset as the processor. The reset port of the bus bridge, `Reset`, is active-high. The AHB reset signal, `HRESETn`, is not an input of the bridge.

If you use synchronous reset while `Reset` is asserted, the AHB-Lite outputs are undefined until the logic driving the output has had 10 positive clock edges. Undefined outputs during this reset period means that spurious requests may be initiated on AHB interface. It is the SoC-level designer's responsibility to mask any ill effects of spurious requests during reset.

If you use asynchronous reset, AHB-Lite outputs will be at their reset (deasserted) state within a definite time after the assertion of the reset signal. The time period needed to reset these outputs is related to the combinational fan-out of the asserting edge of asynchronous reset, which can be ascertained from post-place and route timing reports.

27.4 AHB-Lite Master

This section describes the operation of the AHB-Lite master portion of the bridge.

27.4.1 AHB-Lite Master Port Latency

Xtensa processors support PIF interface option by default. Configuring the AHB-Lite bridge adds one cycle of latency in each direction in comparison with PIF. All of the AHB outputs are driven directly from registers, and most of the AHB inputs are directly registered. A few AHB inputs, such as `HREADY`, are sampled combinationally from the AHB-Lite bus, but the logic's design is to minimize the setup time requirement of these signals. When you select the Asynchronous AMBA Interface option, additional latency is incurred through the asynchronous FIFO's as described in Chapter 26.2.2 "Asynchronous Clock Domains" .

Software simulations of the processor should set the `mlatency` and `blockrepeat` parameter of the instruction-set-simulator to the values specified in Table 27–102.

Table 27–102. Instruction Set Simulator Memory Delay Values[4]

Clock Ratio	`mlatency`	`blockrepeat`
1:1	2 + AHB-latency[1]	PIF-data-width / AHB-transfer-size[2] - 1
2:1	4 + 2 * AHB-latency	2 x PIF-data-width / AHB-transfer-size - 1
3:1	6 + 3 * AHB-latency	3 x PIF-data-width / AHB-transfer-size - 1
4:1	8 + 4 * AHB-latency	4 x PIF-data-width / AHB-transfer-size - 1
Asynchronous	2 + AHB-latency + 3.5 + 3.5 x ratio[3]	ratio x PIF-data-width / AHB-transfer-size - 1

Notes:

1. AHB-latency is the latency to the first word of a read request.
2. Assumes burst read responses have no intra-transfer buswaits.
3. ratio = core frequency / bus frequency
4. The memory delay values are calculated assuming that the processor never flow controls bus responses which may not be true for all Xtensa processors. If the processor does flow control bus responses, the additional cycles will add into the memory delay.

27.4.2 AHB Burst Sizes

Table 27–103 describes the burst sizes used for cache line requests that the processor issues.

Table 27–103. Burst Sizes for Cache Miss Requests

AHB Maximum Transfer Size	Cache Line Size	HBURST
4B	16B	WRAP4
4B	32B	WRAP8
4B	64B	WRAP16
4B	128B	Not supported
4B	256B	Not supported
8B	16B	2 SINGLE
8B	32B	WRAP4
8B	64B	WRAP8
8B	128B	WRAP16
8B	256B	Not supported
16B	16B	SINGLE
16B	32B	2 SINGLE

Table 27–103. Burst Sizes for Cache Miss Requests (continued)

AHB Maximum Transfer Size	Cache Line Size	HBURST
16B	64B	WRAP4
16B	128B	WRAP8
16B	256B	WRAP16

Single data requests are issued for uncached and bypass requests. These request use the burst sizes indicated in Table 27–104. For 8B transfers, two SINGLE requests are issued, as the AHB protocol has no specified provision for a 2-transfer block request.

Table 27–104. Burst Sizes for Single Data PIF Requests

AHB Max Transfer Width	Number Bytes Requested	HBURST
4B	1B	SINGLE
4B	2B	SINGLE
4B	4B	SINGLE
4B	8B	2 SINGLE
4B	16B	WRAP4
8B	1B	SINGLE
8B	2B	SINGLE
8B	4B	SINGLE
8B	8B	SINGLE
8B	16B	2 SINGLE
16B	1B	SINGLE
16B	2B	SINGLE
16B	4B	SINGLE
16B	8B	SINGLE
16B	16B	SINGLE

Note that the PIF-to-AHB-Lite bridge does not issue INCR or INCRx burst size requests on the AHB-Lite bus.

27.4.3 AHB Protection

The values driven on the AHB protection signal by the PIF-to-AHB-Lite bridge depend on the selection of the PIF Request Attributes option while configuring the processor. If this option is not selected, the bridge drives the AHB protection signal HPROT to a con-

stant value of 4’b0011.When the processor is configured with the PIF Request Attributes option, the additional PIF master signal POReqAttribute[11:0] enables the AHB-Lite Master bridge to automatically generate the HPROT protection signal based on the type of the access and attributes of the memory region being accessed. Table 27–105 describes the conversion.

Table 27–105. AHB Protection Signal

HPROT Bit	Description
HPROT[0]	Set to 0 for opcode fetches, set to 1 for data accesses
HPROT[1]	Set to 1 for privileged accesses, set to 0 for user access. Set to 0 for accesses not clearly attributable to a specific instruction, such as castouts or prefetches.
HPROT[2]	Set to 1 for bufferable access, set to 0 for non- bufferable access This bit can be overridden by the instruction (e.g. S32RI, S32NB, S32C1I all set the bufferable bit to 0,).
HPROT[3]	Set to 1 for cacheable access, 0 for non- cacheable access

27.4.4 AHB Errors

AHB errors, encoded in HRESP, are communicated back to the processor. To report errors on write requests, the processor must configure both the write response option, as well as allocate an interrupt of type write-bus error.

27.4.5 AHB Lock

The AHB-Lite bus bridge generates both AHB lock signals, HLOCK and HMASTLOCK for atomic operations. In full AHB systems, an AHB master generates the HLOCK signal, and the arbiter asserts the HMASTLOCK signal to the slaves once it has granted the bus to the requesting AHB master. The bridge generates both lock signals, with a 1-cycle delay from the rising edge of HLOCK to the rising edge of HMASTLOCK.

27.4.6 Locked Requests for Atomic Operations

The Xtensa processors that are configured with the Conditional Store Synchronize Instruction option can perform an atomic update of the memory when an S32C1I instruction is issued.You can use this for implementing synchronization primitives for inter-process communication by providing exclusive access to data (e.g., locks). Executing an S32C1I instruction always results in a read request on AHB. Optionally, a write request may be issued later if the read data matches the value in the SCOMPARE1 register. AHB lock signals HLOCK and HMASTLOCK are used to keep the read-compare-write sequence atomic.

Note: The AHB bridge does not support the exclusive store option and does not work with exclusive load/store instructions.

Figure 27–153 shows an atomic update operation where the data in the memory matches with the value in the SCOMPARE1 register. In cycle 2, HLOCK is first asserted, then in cycle 3, HMASTLOCK is asserted. Most AHB-Lite systems can ignore the HLOCK signal. The read request is issued in cycle 3, and when the read data matches the comparison data, a write request is issued in cycle 7. The lock is deasserted beginning in cycle 8.

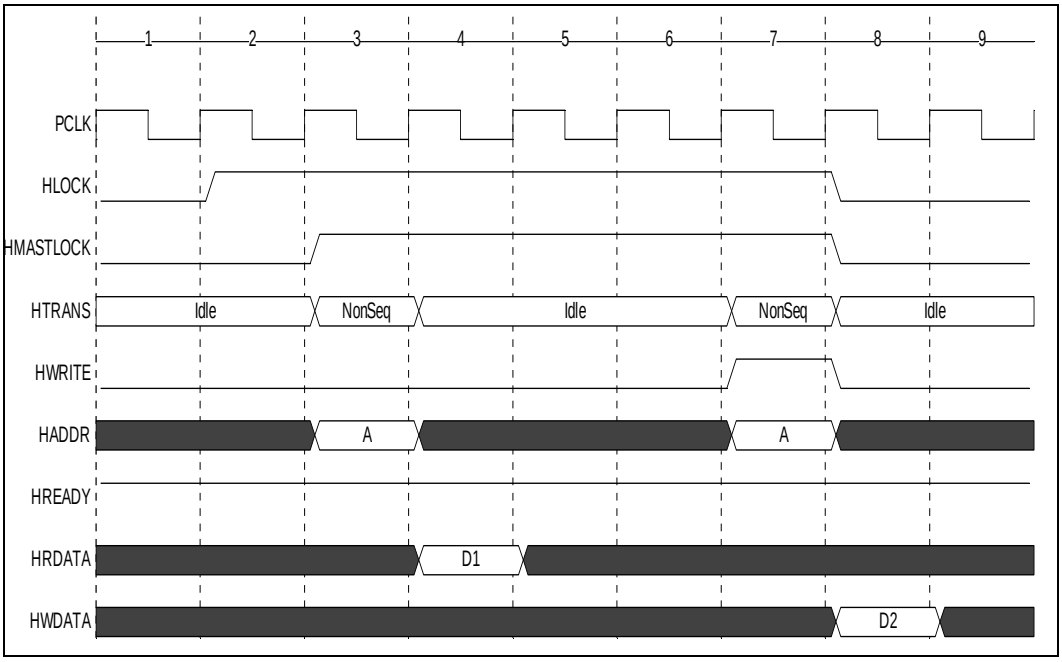


Figure 27–153. AHB Atomic Operation with Match

Figure 27–154 shows a read-conditional-write request where the data does not match. In this case, the locks are deasserted in cycle 6, and no write requests are issued.

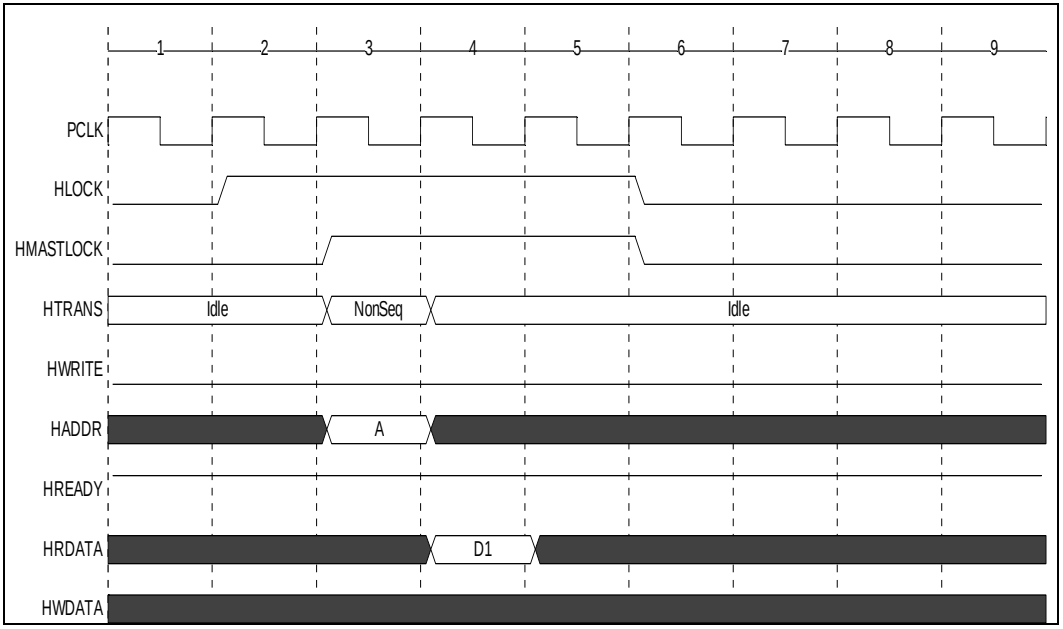


Figure 27–154. AHB Atomic Operation with Miscompare

27.5 AHB-Lite Slave

The optional AHB-Lite slave port implements the subset of the AHB-Lite protocol as defined in this section. Unsupported request types will receive an error response.

27.5.1 AHB Request Queuing

The PIF-to-AHB-Lite bridge does not queue up multiple AHB-Lite requests.

27.5.2 AHB Burst Operations

In general, AHB burst requests of type INCR[4,8,16] and WRAP[4,8,16] that have the starting address aligned to the block size, yield the best performance. All other requests are broken into multiple single accesses. These back-to-back requests are not guaranteed to be atomic, thus they will perform much slower than an aligned burst.

Therefore, we recommend that system designers limit the requests to this subset.

27.5.3 Unspecified Length Bursts

Bursts of an unspecified length will always be treated as unaligned bursts and result in a series of back-to-back requests.

27.5.4 AHB Write Response

The behavior of responses to writes on AHB depends on if the write response option has been configured. For optimal performance incrementing bursts that are aligned to the full burst size and configure the bridge to not to wait for PIF write responses.

- When the write response option is not enabled, there is no mechanism to report errors on writes, thus all writes will receive an OKAY response as soon as the write request has been issued.
- When the write response option is enabled, the bridge is able to communicate any bus errors that are reported by the processor. All transfers of a write burst are accepted on AHB, except for the last transfer. The last response is delayed until the PIF write response is received. There is a 1-cycle latency from the PIF write response to the final write response (as indicated on HREADY).

27.5.5 AHB Protection

The AHB protection control signal, HPROT, is not used by the PIF-to-AHB-Lite bridge.

27.5.6 AHB Lock

The PIF protocol has no support to lock transactions. The PIF-to-AHB-Lite bridge ignores the lock signal.

27.5.7 AHB Transfer Size

The PIF-to-AHB-Lite bridge accepts requests of all transfer sizes, up to the data bus width.

27.5.8 AHB Split and Retry

The PIF-to-AHB-Lite bridge does not implement the split and retry protocol. This is because it is assumed that the local memory access latency will be well bounded.

27.5.9 AHB-Slave Port Latency

The PIF-to-AHB-Lite bridge registers the AHB request outputs and response input in order to help the system designer with timing. This registering incurs a 2-cycle latency on every request. The HREADY and HRESP signals are sampled combinationally on the AHB interface, but the bridge design attempts to sample them as late into the cycle as possible.

27.5.10 The HREADY Connection

Figure 27–155 shows how to connect the slave HREADY_OUT_S and HREADY_IN_S signals. For each AHB-Lite layer, there is one HREADY signal, which is generated by muxing the HREADY_OUT_S signals of all the slaves in the layer. This HREADY signal goes into an input port of the masters and slaves in the layer.

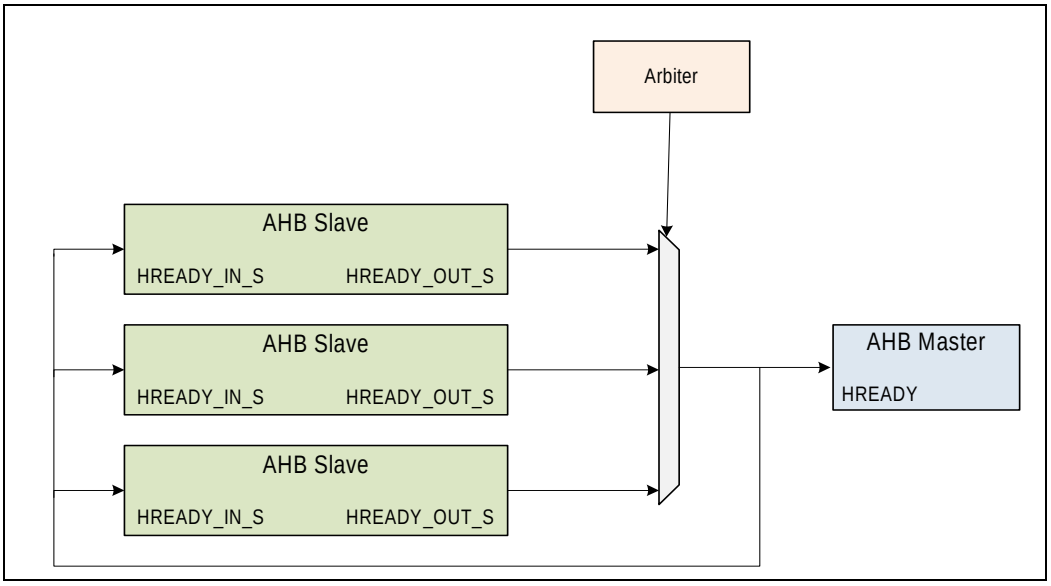


Figure 27–155. AHB Slave **HREADY** Connectivity

Note that the AHB specification requires HREADY to always be asserted during the data phase of IDLE transfers. This means that the first address phase of a new request cannot be extended by deasserting HREADY, except as a side-effect of extending the data phase of a previous (non-IDLE) request.

27.6 Read Requests

This sections shows the example timing of several different types of read requests. The bridge is optimized for aligned burst read requests as in Section 27.6.2.

27.6.1 Read Single

Figure 27–156 shows the timing of a single data read received on the AHB-Lite slave port. The request is received in cycle 2 and registered. It is then issued on the PIF in the next cycle. The example shows the fastest response that 5-stage Xtensa processors will give; the processor issues the request in it its local memory in cycle 5, and the first response data can be received in cycle 6, which is then registered onto the PIF in cycle 7. The bridge registers all of its outputs, sending the final response to the AHB request in cycle 8. There is a minimum of 6 cycles of latency, as shown in the example. Note that a 7-stage configuration has an additional cycle of latency.

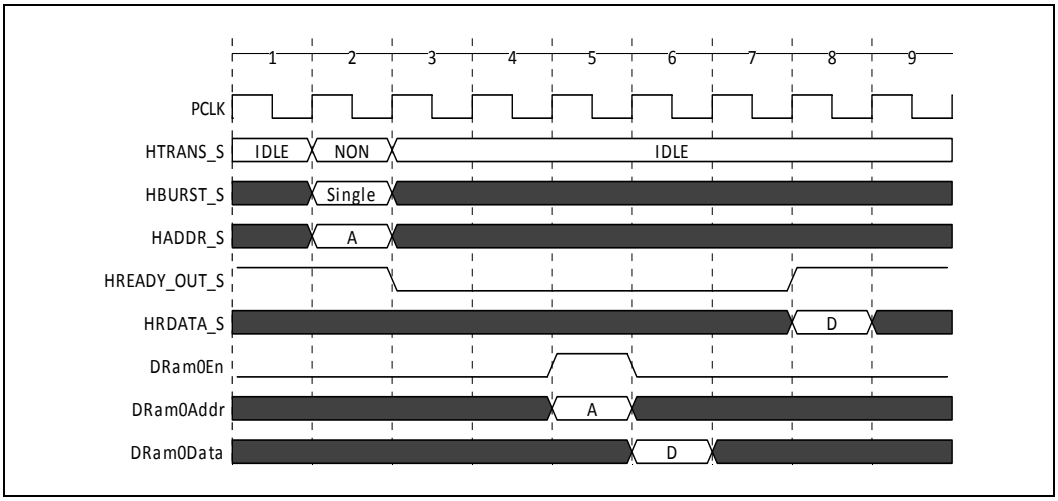


Figure 27–156. Inbound Read Single Example

27.6.2 Aligned Read Burst

Figure 27–157 shows the timing of a burst read received on the AHB-Lite slave port. This example is similar to the single data read example, except there are four response transfers instead of one. This example assumes that the address is well aligned as explained in Section 27.5.2 “AHB Burst Operations”.

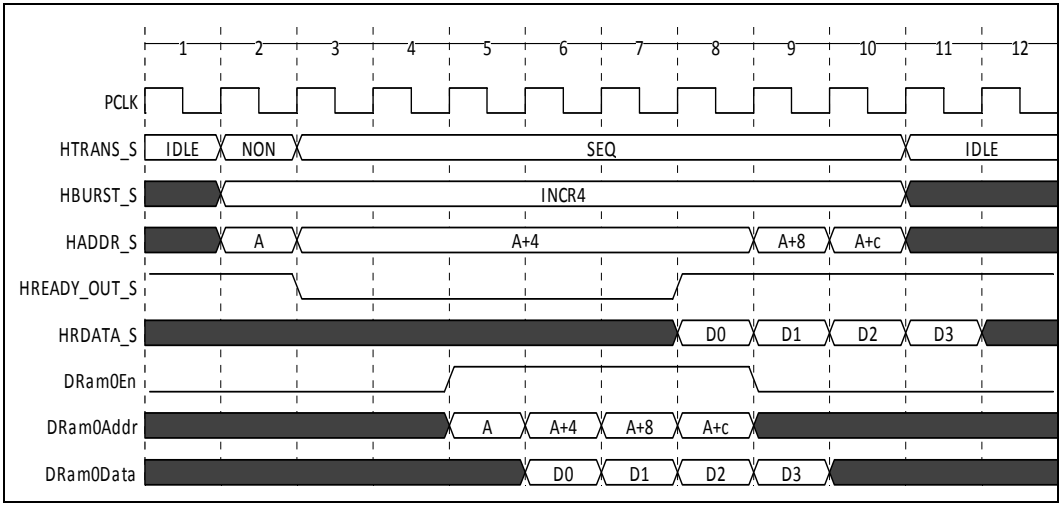


Figure 27–157. Inbound Aligned Read Burst Example

27.6.3 Unaligned Read Burst

Figure 27–158 shows an unaligned burst read example. Unaligned bursts are handled as if they were a series of back-to-back single data requests. The first PIF request in cycle 3 is for a single transfer, rather than an entire block of four. The second PIF request is issued in cycle 9, after the first transfer has completed on the AHB-Lite bus.

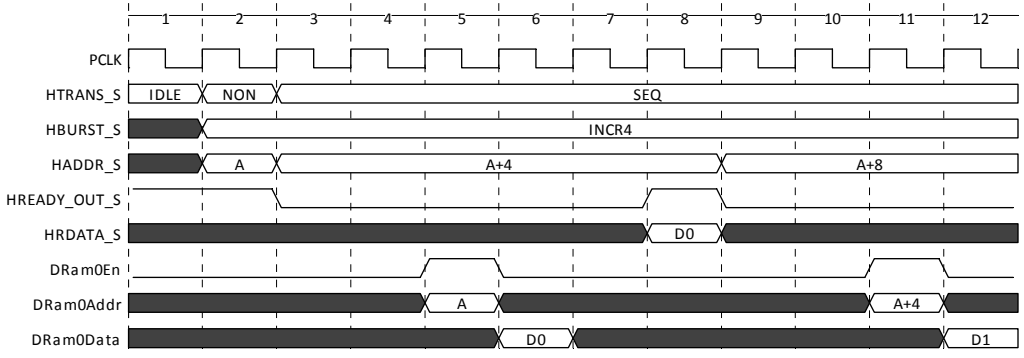


Figure 27–158. Inbound Unaligned Read Burst Example

27.7 Write Requests

This section shows examples of inbound write operations. The bridge is optimized for aligned burst requests as described in Section 27.5.2 “AHB Burst Operations”. The example in Section 27.7.1 shows an aligned burst write.

27.7.1 Aligned Burst Write

Figure 27–159 shows an example of a burst write request that is well aligned. A PIF block write request is issued in cycles 4 through 7. Assuming that the Xtensa core is not configured with PIF write response, the last data phase is extended by two cycles in order to wait for the last transfer to complete on the PIF. If PIF write response is configured, the bridge must wait for an extra cycle at least, to receive the write response acknowledgment from PIF.

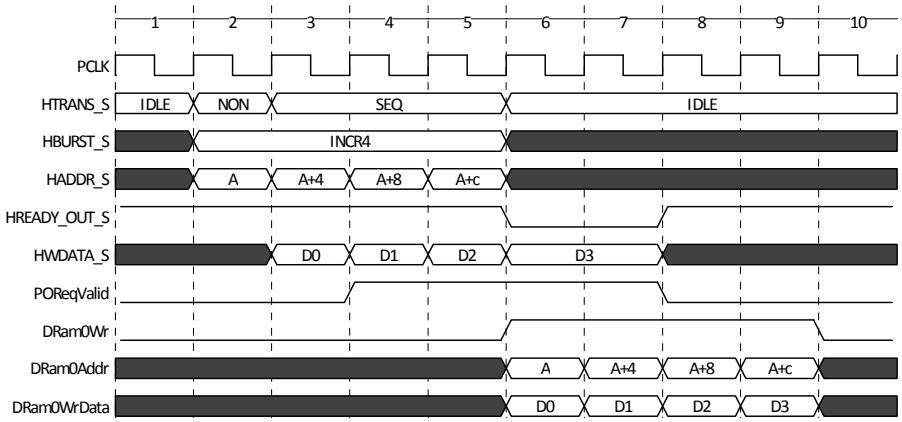


Figure 27–159. Inbound Aligned Burst Write

27.7.2 Unaligned Burst Write

Figure 27–160 shows an example of an unaligned burst write request assuming PIF write response is not configured. Each transfer of the write request is flow controlled and handled as though it were a series of back-to-back write requests.

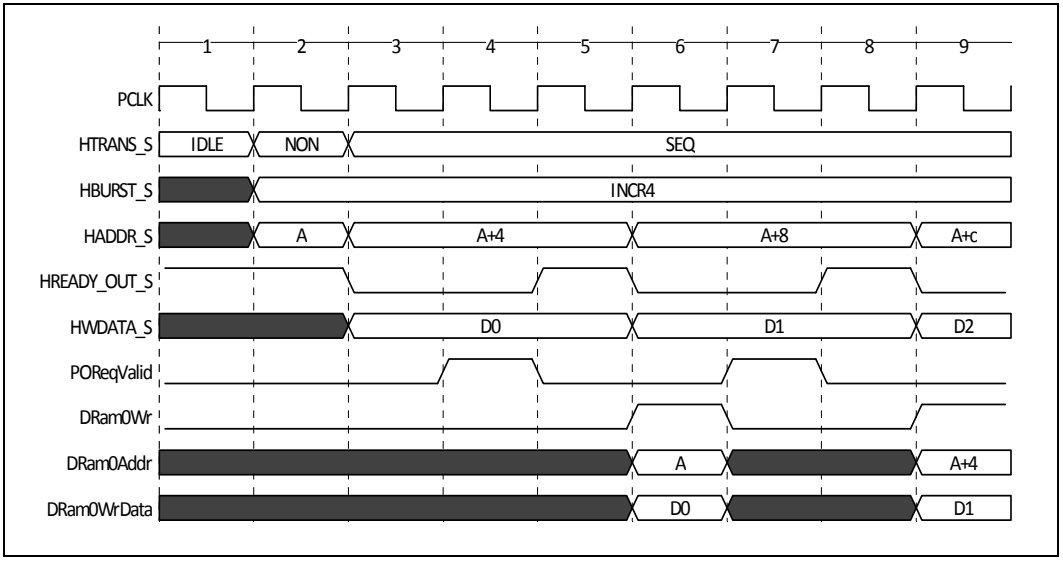


Figure 27–160. Inbound Unaligned Burst Write

28. AXI and ACE-Lite Bus Bridges

The PIF-to-AXI bridge allows designers to integrate Xtensa processors into AMBA[1] AXI3, AXI4 or ACE systems. The PIF-to-AXI bridge can be configured with the AXI3 option to be compatible with the original AMBA AXI 1.0 specification. Alternatively it can be configured with the AXI4 option to be compatible with an AXI4 bus as specified in version 2.0 of the AMBA AXI protocol specification. Optionally the AXI4 bridge can also support an ACE-Lite interface which allows it to be integrated into ACE systems.The ACE-Lite option allows a Xtensa processor to snoop external ACE masters, however the external ACE masters can not snoop Xtensa's caches, thus providing *one-way* cache coherence.

28.1 Separate AXI Master Port for iDMA

The integrated DMA (iDMA) requires the AXI bus option to be selected. When the iDMA is configured, the Xtensa processor provides two separate AXI master ports. One AXI master port services outbound processor requests for cache fills, dirty-line castouts, prefetches, and uncached requests. The other AXI master port only services the requests originating from the iDMA. If the iDMA is not configured, only one AXI master port exists, which services all processor requests. See Chapter 19 for information about iDMA.

Both AXI master ports have identical configurations (for example, data bus width, clocking, etc.) The only functional difference between the two is that the read request originating from iDMA is strongly ordered by the re-use of ID. Additionally, the iDMA can use the priority information in the descriptor to drive the QOS signals in the AXI4 bridge. For the processor AXI4 port, the QOS signals are fixed.

Xtensa also provides an option to configure an AXI slave port that gives other AXI masters access to the processor's local memories. It is useful for initialization through external DMA or high-performance data movement without suffering cache latency.

28.2 Configuration Options

The PIF-to-AXI bridge supports the following configuration options.

- Processor Data Bus Width
  - 32-, 64- and 128-bit processor read and write data bus widths are supported

1. AMBA and ARM are registered trademarks of ARM Limited. AMBA (Advanced Microcontroller Bus Architecture) is an open-standard specification of a set of on chip buses and interconnects. AXI (Advanced eXtensible Interface) is one of the protocols defined within the AMBA specification.

- Cache Line Size
  - 16-, 32-, and 64-byte cache lines are supported for all configurations
  - 128-byte cache lines are supported for configurations with 64- or 128-bit data bus widths
  - 256-byte cache lines are supported for configurations with 128-bit read data bus widths
- AXI Bridge Type
  - AXI3,which provides a PIF-to-AXI bridge with the AXI interface compatible with AXI3 bus
  - AXI4,which provides a PIF-to-AXI bridge with the AXI interface compatible with AXI4 bus
- ACE-Lite option: This interface allows an Xtensa processor to be integrated with an ACE interconnect. The ACE-Lite option is only supported with AXI4 and not with AXI3.
- AXI-Slave: Use the optional AXI-slave port to access the processor's local memories directly from an external AXI master, such as a DMA controller. The Xtensa processor is capable of executing from an instruction or data memory without stalling while a different instruction or data memory is being accessed through the bridge's AXI-slave port.
  - AXI-Slave Buffer Depths: The optional AXI-slave port converts AXI requests to PIF requests and PIF responses back to AXI responses. Internally, it buffers the requests and responses. The depths of these buffers can have a significant impact on area and performance, thus the depths of these buffers are offered as configuration options via the Xtensa Processor Generator. A system designer should set them based on processor and bus clock speeds, the nature of the bus traffic, and system specific requirements.The depths of the following AXI-slave buffers is configurable:
  - The AXI slave Request Control Buffer Depth can be set to one of 1,2,4,8,16 entries. The default is 4 entries. Each entry consists of the AXI request control signals used by the bridge viz address, length, size, burst type and ID bits. This selection applies to both Read Address and Write Address channels.
  - The AXI slave Request Data Buffer Depth can be set to one of 1,2,4,8,16 entries. The default is 8 entries. Each entry consists of AXI write request data signals used by the bridge viz data, strobes, and ID bits. This selection only applies to Write Data channels.
  - The AXI slave Response Buffer Depth can be set to one of 1,2,4,8, or 16 entries. The default is 8 entries. This parameter changes the number of entries in the read response buffer. Each entry in the read response buffer stores read data bits along with error control and ID bits. Note that the selection of this parameter only applies to the Read Response channel. The depth of the AXI slave write response buffer, which is not configurable, is set to 16 entries.

PIF Request Attributes: The PIF interface of the processor can be configured to use PIF Request Attributes. It adds an additional Request Attribute output for a PIF master (POReqAttribute[11:0]), and input for a PIF slave (PIReqAttribute[11:0]). When the PIF Request Attributes option is selected, the PIF-to-AXI bridge adds two extra ports, ARXTUSER[3:0] and AWXTUSER[3:0] to the AXI master interface. The bottom four bits of the POReqAttribute[11:0] signal are assigned to ARXTUSER[3:0] for read requests and to AWXTUSER[3:0] for write requests. The PIF-to-AXI bridge also uses the top eight bits of the POReqAttribute[11:0] signal to drive ARCACHE[3:0], AWCACHE[3:0], ARPROT[2:0] and AWPROT[2:0] signals on the AXI bus. For details, refer to Section 28.6.11 “AXI Master Cache Control” and Section 28.6.13 “AXI Master Protection Control”. If PIF Request Attributes are not configured, all four of the above mentioned signals are driven to 0.The PIF Request Attribute option does not make any changes to the AXI slave interface.

- Critical Word First
  - When the processor is configured with the Critical Word First option, it may send out WRAP requests on the AXI bus with the starting address not aligned to the burst size.
  - If the Critical Word First option is not selected, all AXI burst requests will be issued on the AXI bus with start addresses aligned to the burst size.
- Arbitrary Byte Enables
  - A PIF master can use the POReqDataBE signal to indicate the byte lanes that contain valid data, allowing sparse data transfer on the read/write data buses. The PIF 4.0 protocol allows an Xtensa processor to be configured with the Arbitrary Byte Enables option,which allows any arbitrary bit pattern on the POReqDataBE signal of the PIF master interface. Without this option only a few bit combinations are legal in the POReqDataBE signal. (Refer to Section 13.2.2 “Optional Arbitrary Byte Enables” for more details.)
  - If the processor is configured with the Arbitrary Byte Enables option, the arbitrary bit patterns that may appear on the POReqDataBE signal are passed on to the WSTRB signal of the AXI master interface.
  - Note that for an Xtensa processor that implements PIF4.0 protoco, te Arbitrary Byte Enables option is enabled by default on its inbound PIF port. This ensures improved performance when handling inbound AXI write requests through the PIF-to-AXI slave port.
- AXI security: This option provides a way to mark the outbound AXI master transactions as secure or non-secure. It can also be used to make the AXI slave port security aware by blocking incoming AXI transactions that are not secure.
- Parity/ECC protection: The PIF-to-AXI bridge can be configured with an error protection layer for the AXI signals. It uses ECC and Parity codes to protect data and control signals respectively.

Exclusive Store option: This option provides ISA instructions that allow initiating exclusive AXI transactions on the Master port. When this option is selected, the AXI slave port supports exclusive transactions as well. See Section 28.10.8 “AXI Slave Exclusive Accesses” for details. Note that this option is only supported with AXI4.

- Clocking Mode: Synchronous and asynchronous modes are available.

28.3 AXI Specification Compliance

You can select one of the two different versions of the PIF-to-AXI bridge (compatible with AXI3 and AXI4 standards, respectively) while configuring the bridge.

28.3.1 AXI3 Standard Compliance

The AXI3 PIF-to-AXI bridge is compliant with the AXI3 protocol. The implementation of the AXI3 master performs a subset of the AXI protocol that maps to the requests that the Xtensa processor is capable of producing. The AXI3 master does not interleave data from different write requests.

The AXI3 slave port can receive all types of AXI requests, however request performance depends on how well the AXI request maps to a PIF request. The AXI3 slave bridge has an interleaving depth of 1, which means streams of different writes cannot be interleaved. The AXI3 slave bridge does not perform locked or exclusive operations. The slave bridge does not use AXI cache control signals. If the AXI security option is configured, the secure bit in AXI protection signals is used, otherwise the protection signals are ignored.

28.3.2 AXI4 Standard Compliance

The AXI4 PIF-to-AXI bridge is compliant with the AXI4 protocol. It has a different pinout compared to the AXI3 bridge offering wider ARLEN and AWLEN signals and shorter ARLOCK and AWLOCK signals. The AXI4 PIF-to-AXI bridge also supports Quality of Service (QOS) signals as specified by the AXI4 standard. Since interleaving of different writes is not supported under the AXI4 standard, the AXI4 bridge removes WID signals from both AXI master and slave interfaces. Similar to the AXI3 master, the AXI4 master only performs a subset of the AXI protocol that maps to PIF requests that the Xtensa processor is capable of producing. However, the AXI4 master implements atomic access requests in a different way when compared with the AXI3 master.

Enabling the ACE-Lite option adds additional signals to read and write address channels as defined by the standard.Note that the Xtensa processor never generates ACE Cache Maintenance operations, Barrier transactions, or Distributed Virtual Memory transactions.

The AXI4 slave port can receive all types of AXI requests, including the longer bursts (up to 256 beats) specified in the AXI4 standard. Request performance depends on how well the AXI request maps to a PIF request. The AXI4 slave bridge supports exclusive transactions if the Exclusive Store option is selected. The slave bridge does not use AXI cache and protection control interfaces.

Both AXI4 the master and slave bridges do not support use of ARREGION and AWREGION signals.

- For more information about the AXI protocol, refer to the AMBA™ AXI and ACE Protocol Specification (rev 2.0) ARM IHI 0022E, available from ARM.

28.4 Reset

The PIF-to-AXI bridge is reset with the same active-high reset as the processor. The reset port, `Reset`, is active-high. The AXI reset signal, `ARESETn`, is not an input of the bridge.

If you use Synchronous Reset while `Reset` is asserted, the outputs of the PIF-to-AXI bridge are undefined until the logic driving the output has had 10 positive clock edges. Undefined outputs during this reset period means that spurious requests may be initiated on the AXI interface. It is the SoC-level designer's responsibility to mask any ill effects of spurious requests during reset.

If you use asynchronous reset, AXI bridge outputs will be at their reset (deasserted) state within a definite time after the assertion of the reset signal. The time period needed to reset these outputs is related to the combinational fan-out of the asserting edge of asynchronous reset, which can be ascertained from post-place and route timing reports.

28.5 Byte Invariance

AXI is defined as a "byte-invariant" interface, such that all transactions must follow little-endian byte ordering. The PIF-to-AXI bridge performs byte-lane reversals for big-endian processor configurations.

Note that byte transfers between little-endian and big-endian processors and devices are naturally supported in a byte-invariant scheme, but all wider transfers may require an additional byte reversal. For example, to transfer a 32-bit integer word between a little endian- and big-endian processor, either the producer or consumer would have to perform a byte reversal in software to undo the byte reversal that would occur in the bridge of the big-endian processor.

28.6 AXI Master

This section describes the AXI protocol implemented by the PIF-to-AXI Master bridge. The description is generally applicable to all the bridge types, AXI3, AXI4 and ACE-Lite, unless explicitly mentioned.

28.6.1 AXI Master Port Latency

All PIF-to-AXI bridges add one cycle of latency to read requests and two cycles to write requests.

Selecting the Asynchronous AMBA Bridge Interface option adds 3.5 cycles of core clock and 3.5 cycles of bus clock latency, as described in Chapter 26.2.2 “Asynchronous Clock Domains” .

28.6.2 AXI Master Read and Write Channel Ordering

AXI has separate physical channels for read and write requests. The ordering between reads and writes issued by Xtensa processors is determined based on configuration-dependent ordering rules. The bridge operation depends on if the Write Response option is selected.When the Write Responses option is selected, the bridge attempts to make use of the higher bandwidth (offered by separate read and write channels), by allowing read and write requests to be issued in parallel. However, the bridge is careful to detect potential read-after-write hazards. That is, it is assumed that a read will always see the effects of the previous writes to the same address. Because AXI has separate write and read channels, the order of requests on AXI may be different than the order of requests issued by the processor, potentially allowing a younger read to be performed ahead of an older write. To remedy this reordering hazard, the bridge will delay read requests from being issued whenever there are pending or outstanding writes that have overlapping addresses. The address overlap is computed by comparing 8-12 bits of address. The exact number of bits being compared and their position depends on the cache line and data bus width as defined in Table 28–106. Not comparing the full 32-bit address may result in some false matches, which does not affect correct functionality, but does save the area required to store address.

Table 28–106. Address Bits used to Check for Read-After-Write Hazard

Data Cache Configured	Data Cache Line SIze	Data bus Width	Address Bits Compared	No. of Address Bits Compared
No	n/a	32-bit	[13:2]	12
No	n/a	64-bit	[14:3]	12
No	n/a	128-bit	[15:4]	12
Yes	16B	32-bit	[13:4]	10

Table 28–106. Address Bits used to Check for Read-After-Write Hazard (continued)

Data Cache Configured	Data Cache Line SIze	Data bus Width	Address Bits Compared	No. of Address Bits Compared
Yes	16B	64-bit	[14:4]	11
Yes	16B	128-bit	[15:4]	12
Yes	32B	32-bit	[13:5]	9
Yes	32B	64-bit	[14:5]	10
Yes	32B	128-bit	[15:5]	11
Yes	64B	32-bit	[13:6]	8
Yes	64B	64-bit	[14:6]	9
Yes	64B	128-bit	[15:6]	10
Yes	128B	64-bit	[14:7]	8
Yes	128B	128-bit	[15:7]	9
Yes	256B	128-bit	[15:8]	8

When write responses are not selected, the bridge implements a more conservative ordering model, which prevents reads from being issued until all outstanding and pending writes have completed.

28.6.3 AXI Master Write Request Ordering

The AXI master may issue multiple outstanding write requests, but the AXI protocol does not guarantee that write requests will complete in order unless the requests use the same ID. Hence, the AXI master needs to carefully detect potential write-after-write hazard cases to ensure they are correctly ordered. To detect a write-after-write hazard, the bridge checks if the current write request overlaps with any of the outstanding write requests. The address comparison for overlap always uses eight bits; the position of the bits depends on the configured data bus width as defined in Table 28–107. If overlap is detected, the bridge reuses the ID of the outstanding write request for the current write request. Note that the write-after-write hazard comparison works the same way whether write responses are selected or not selected.

Table 28–107. Address Bits used to Check for Write-After-Write Hazard

Data Bus Width	Address Bits Compared	No. of Address Bits Compared
32-bit	[13:6]	8
64-bit	[14:7]	8
128-bit	[15:8]	8

28.6.4 AXI Master Write Address and Data Channel Synchronization

When all previous write requests have been flushed, the AXI bridge will always initiate the new write request simultaneously on the write address and write data channels, by asserting `AWVALID` and `WVALID` in the same cycle at the start of a write request. However from that point onwards, write address and write data channels may operate at different rates based on the how the AXI bus flow controls them.

28.6.5 AXI Master Multiple Outstanding Requests

The AXI master bridge generates multiple outstanding AXI requests. The read channel may issue up to 16 outstanding read requests on the AXI bus. The write channel can also issue up to 16 outstanding write requests on the AXI bus.

If iDMA is configured, a second AXI Master port is added to the design. The maximum number of outstanding requests generated by this port is equal to the value programmed in the iDMA settings register. Section 19.3.2 “Settings Register ” on page 354 contains details about this register.

28.6.6 AXI Master Request ID Usage

The AXI master port servicing processor requests may use up to 16 different request IDs (0 to 15) for AXI read requests. New reads will never use the ID of an outstanding read request. The responses for different outstanding read requests are allowed to return interleaved, the AXI master is capable of handling them.

If iDMA is configured, a second AXI Master port is added to the design. This port uses the same ID for all AXI read requests and the response is expected to come back in order and non-interleaved.

The AXI master port servicing processor requests may also use up to 16 different request IDs (0 to 15) for AXI write requests. The issuing of new write requests may be stalled if all 16 IDs are in use. Writes that have overlapping bytes with outstanding writes will reuse the ID of the outstanding write, to guarantee that the writes are processed in order. The usage of write IDs change based on selection of PIF Request Attributes configuration option.

Write ID Usage with PIF Request Attributes Configured

If PIF Request Attributes are configured, the PIF master drives an additional POReqAttribute[11:0] signal. Bits POReqAttribute[8:5] provide the cache attribute information associated with the request. Bit POReqAttribute[5] indicates the Cacheable attribute, bit

POReqAttribute[6] indicates the Writeback attribute, bit POReqAttribute[7] indicates the Read Allocate attribute and bit POReqAttribute[8] indicates the Write Allocate attribute. Refer to Table 28–108 for more details.

The AXI master uses the PIF cache attribute information to strongly order bypass and write-through write requests. Below is a list of write ID usage rules that the AXI master follows when PIF Request Attributes are configured:

- All bypass or write through (POReqAttribute[8:5] == 4'bxx01) write requests use an AXI write ID of 0.
- If a bypass or write through (POReqAttribute[8:5] == 4'bxx01) write request overlaps with an outstanding write request that has a non-zero ID, it stalls until the outstanding write request has completed.
- All write-back (POReqAttribute[8:5] == 4'bxx11) write requests use any AXI write ID from 1 to 15, unless they overlap with an outstanding bypass or write-through write request.
- If a write-back (POReqAttribute[8:5] == 4'bxx11) write request overlaps with an outstanding bypass or write-through write request, it uses an AXI write ID of 0.
- If a write-back (POReqAttribute[8:5] == 4'bxx11) write request overlaps with an outstanding write-back write request, it reuses the AXI write ID of the outstanding write request.

Write ID Usage Without PIF Request Attributes Configured

If PIF Request Attributes are not configured, the POReqAttribute[11:0] signal does not exist and the cache attribute information is unavailable. All AXI write requests may use any request IDs between 0 to 15. Writes that have overlapping bytes with outstanding writes will reuse the ID of the outstanding write to prevent write-after-write hazard.

Note: The iDMA AXI Master port always uses the same ID for all AXI write requests.

28.6.7 AXI Master Burst Length

The processor AXI Master port and iDMA AXI Master port (if iDMA is configured) generate requests with burst lengths of 1, 2, 4, 8, or 16 transfers only. Note that even though the AXI4 standard allows requests with a burst length greater than 16, the AXI4 master does not generate these longer burst requests.

28.6.8 AXI Master Burst Size and Write Strobes

The AXI bridges generate all bursts greater than one transfer with the size set to the complete width of the data bus. Requests of one transfer may be for partial bus widths, for example a byte wide request will have a size of "byte".

If PIF Arbitrary Byte Enables are configured, the AXI master can generate write requests where the request address is lower than, or equal to the first enabled byte and the WSTRB signal can have sparse strobes with a non-power of two bytes enabled. Using a 64-bit bus as an example, the write request may have an address of 0x0 and write strobes as 8'b0010110. Sparse strobes will always have a size set to the complete width of the data bus.

If PIF Arbitrary Byte Enables are not configured, or if no TIE store operations have been written to make use of them, write strobes are always consistent with the size field. For example, a request for a half-word will always have two strobes enabled that are aligned to the requested address.

Only the AXI3 master may issue requests with all strobes disabled as a part of an atomic operation to release a lock.

Note: The requests generated by the iDMA AXI Master may use sparse strobes regardless of the burst length.

28.6.9 AXI Master Burst Type

Requests greater than one transfer will be issued as bursts of type of WRAP. All other requests will be issued with a burst type of INCR.

28.6.10 AXI Master Burst Addresses

During AXI requests greater than one transfer, the AXI master always drives the ARSIZE and AWSIZE field to indicate the full width of the bus, and always aligns the address to the full width of the bus. If the Critical Word First option is not selected, all AXI burst operations have a start address that is aligned to the transfer size * length (i.e., the total number of bytes in the burst). That is, without Critical Word First, the start address points to the transfer that has the lowest address. If the Critical Word First option is chosen, the start address may point to any transfer of the burst. Both read and write requests of only one transfer will always have addresses lower than or equal to the first enabled byte in the transfer, and ARSIZE, AWSIZE will be set appropriately.

Note: The iDMA AXI Master port always sends out requests with the starting address aligned to the total number of bytes in the burst.

28.6.11 AXI Master Cache Control

The AXI cache interface is only used by the bridge when the PIF Request Attributes option is selected, otherwise cache signals ARCACHE and AWCACHE are driven to 0. If the PIF Request Attributes option is selected, the bridge use the Xtensa processor's

cache attributes for the addressed region to drive ARCACHE and AWCACHE signals. Table 28–108 describes the AXI3 bridge ARCACHE and AWCACHE values as a function of Cache Attributes.

Table 28–108. ARCACHE and AWCACHE Values for AXI3 Master Cache Attributes

Processor Cache Attribute	AXI3 Cache Attribute	W-Allocate 3	R-Allocate 2	Cacheable 1	Bufferable 0
Bypass	Noncacheable and nonbufferable	0	0	0	0
Bypass bufferable	Bufferable only	0	0	0	b
Cached no-allocate	Cacheable but do not allocate. Bufferable unless overridden and set to 0	0	0	1	b
Write-through	Cacheable write-through allocate on reads only	0	1	1	0
Write-back no-write-allocate	Cacheable write-back allocate on reads only	0	1	1	1
Write-back write-allocate	Cacheable write-back allocate on both reads and writes	1	1	1	1
Note: "b" typically means 1, unless overridden and set to 0 by an instruction such as S32NB, L32AI, S32RI, and S32C1I.					

The ARCACHE and AWCACHE values for the AXI4 bridge depend on the selection of the memory management unit. If the Memory Protection Unit (MPU) is configured, the values are as described in Table 28–109. Otherwise the values are as described in Table 28–110

Table 28–109. AXI4 Master with MPU: Cache Attributes

Processor Memory Type	AXI4 Cache Attributes	ARCACHE Bits				AWCACHE Bits			
		O Allocate 3	Allocate 2	Modifiable 1	Bufferable 0	Allocate 3	O Allocate 2	Modifiable 1	Bufferable 0
Device non-bufferable	Device non-bufferable	0	0	0	0	0	0	0	0
Device bufferable	Device bufferable	0	0	0	b	0	0	0	b
Non-cacheable non-bufferable	Non-cacheable non-bufferable	0	0	1	0	0	0	1	0
Non-cacheable bufferable	Non-cacheable bufferable	0	0	1	b	0	0	1	b
Cacheable	Write-through no allocate	1	0	1	0	0	1	1	0
Cacheable	Write-through read allocate	1	1	1	0	0	1	1	0
Cacheable	Write-through write allocate	1	0	1	0	1	1	1	0
Cacheable	Write-through read and write allocate	1	1	1	0	1	1	1	0
Cacheable	Write back no allocate	1	0	1	1	0	1	1	1
Cacheable	Write back read allocate	1	1	1	1	0	1	1	1
Cacheable	Write back write allocate	1	0	1	1	1	1	1	1
Cacheable	Write back read and write allocate	1	1	1	1	1	1	1	1

Note:

Processor memory type Cacheable allows the user to program the cacheability behavior inside the processor as well as the cache attribute that is used on the AXI bus. Refer to Section 3.3.3 “Memory Type[8:0] Field Codes” for details.

"b" typically means 1, unless overridden and set to 0 by an instruction such as S32NB, L32AI, S32RI, and S32C1I.

Table 28–110. AXI4 Master without MPU: Cache Attributes

Processor Cache Attributes	AXI4 Cache Attributes	ARCACHE Bits				AWCACHE Bits			
		O Allocate 3	Allocate 2	Modifiable 1	Bufferable 0	Allocate 3	O Allocate 2	Modifiable 1	Bufferable 0
Bypass	Device non-bufferable	0	0	0	0	0	0	0	0
Bypass bufferable	Device bufferable	0	0	0	b	0	0	0	b
Cached no-allocate	Write-through no allocate	1	0	1	0	0	1	1	0
Write-through	Write-through allocate on reads only	1	1	1	0	0	1	1	0
Write-back no write-alloc	Write-back allocate on reads only	1	1	1	1	0	1	1	1
Write-back write-allocate	Write-back allocate on both reads and writes	1	1	1	1	1	1	1	1
Note: "b" typically means 1, unless overridden and set to 0 by an instruction such as S32NB, L32AI, S32RI, and S32C1I.									

Note that the AXI3 bridge issues all atomic operations with non-cacheable, non-bufferable (where ARCACHE and AWCACHE signals are set to 4'b0000) cache attributes. AXI4 bridge issues all atomic operations with normal non-cacheable, non-bufferable (where ARCACHE and AWCACHE signals are set to 4'b0010) cache attributes.

The iDMA AXI Master port also uses the processor cache attributes to drive AXI cache attributes.

28.6.12 ACE Operations

If the ACE-Lite option is configured, the AXI4 bridge can be integrated into an ACE subsystems. The processor AXI Master bridge and the iDMA AXI Master bridge use the cache attributes and the sharable attribute for the addressed region to issue ACE transactions. Following is a summary of the ACE operations issued by AXI Master ports. Table Table 28–111 below provides the summary for all the read operations and table Table 28–112 provides the summary for all the write operations. Note that the cache attribute signals are driven in the same way as the AXI4 configuration as described in the previous section Table 28.6.11.

Table 28–111. ACE Read Operations Summary

Processor Operation	Sharable?	ACE Operation	ARSNOOP Bits				ARDOMAIN Bits	
			3	2	1	0	1	0
Device Read	Yes	ReadNoSnoop	0	0	0	0	1	1
Memory Read	Yes, inner sharable	ReadOnce	0	0	0	0	0	1
Memory Read	Yes, outer sharable	ReadOnce	0	0	0	0	1	0
Memory Read	No	ReadNoSnoop	0	0	0	0	0	0

Table 28–112. ACE Write Operations Summary

Processor Operation	Sharable?	ACE Operation	AWSNOOP Bits			AWDOMAIN Bits	
			2	1	0	1	0
Device Write	Yes	WriteNoSnoop	0	0	0	1	1
Dirty Line Castout	No	WriteNoSnoop	0	0	0	0	0
Store Operation	Yes, inner sharable	WriteUnique	0	0	0	0	1
Store Operation	Yes, outer sharable	WriteUnique	0	0	0	1	0
Store Operation	No	WriteNoSnoop	0	0	0	0	0

28.6.13 AXI Master Protection Control

The AXI protection interfaces are only used by the bridge when the PIF Request Attributes option is selected. Table 28–113 provides a summary of how ARPROT and AWPROT values are set for the processor AXI Master port and the iDMA AXI Master port. If the PIF Request Attributes option is not selected, protection signals ARPROT and AWPROT are driven to 0.

Table 28–113. AXI Master: Protection Signals

ARPROT and AWPROT Bit	Processor AXI Master Port	iDMA AXI Master Port
Bit 0, privilege bit	Set to 1 for privileged accesses, set to 0 for user access. Set to 0 for accesses not clearly attributable to a specific instruction, such as castouts or prefetches.	Set to 1 for privilege descriptor. Set to 0 for user descriptor
Bit 1, secure bit. 0 indicates secure access, 1 indicates non-secure access.	If AXI Security feature is configured this bit is set based on primary input signal AXISECMST. See Section 28.6.14 for details. Set to non-secure (1'b1) if the AXI Security feature is not configured.	Same as the processor AXI Master port.
Bit2, instruction bit. 1 indicates instruction access, 0 indicates data access	Set to 1 for opcode fetches. Set to 0 for data accesses	Fixed to value 0, all transactions are data accesses.

28.6.14 AXI Master Security Feature

If the AXI Security feature is configured, a primary input, AXISECMST, is added to the Xtensa processor. This port drives the secure bit in AXI security signals (A[R/W]PROT[1]) and is set to be inverse of AXISECMST (~AXISECMST). Note that the state of AXISECMST should only be toggled when there are no pending/outstanding transactions buffered in the processor. One way to ensure this is to execute the WAITI instruction and wait until the primary output signal PWaitMode is asserted.

Note: The secure bits in the security signals of the iDMA AXI Master port (A[R/W]PROT_iDMA[1]) are also set to be inverse of AXISECMST (~AXISECMST).

28.6.15 AXI4 Master Quality of Service

The AXI4 standard extends the master interface to add two 4-bit quality of service identifiers called ARQOS and AWQOS. The Xtensa processor drives these signals to a fixed value of 4'b1000. Note that Quality of Service signals only exist for the AXI4 bridge and not for the AXI3 bridge.

The Quality of Service signals for the iDMA AXI Master port are driven based on the priority setting in the descriptor.

28.6.16 AXI4 Master Multiple Region Interfaces

The AXI4 master does not support the optional multiple region interfaces defined by the AXI4 standard and does not provide ARREGION and AWREGION signals.

28.6.17 AXI Master Error Response Signaling

The AXI protocol allows error response signaling using RRESP and BRESP signals. The error response can be encoded as a slave error (SLVERR) or a decode error (DECERR). The SLVERR response generally indicates an unsuccessful transaction due to an error condition encountered by the slave. The DECERR response is generated when no slave responds to a transaction address.

The Xtensa processor does not distinguish between these two different AXI errors. Both SLVERR and DECERR are treated as data bus errors. AXI errors on the iDMA master port are reported in the iDMA status register.

If AXI Parity/ECC protection is configured, the AXI master logic detects transmission faults on all incoming AXI control and data signals. If a parity error occurs on an AXI control signal or if an uncorrectable ECC error is seen on the read data, the AXI master generates a data bus error response. This may cause the processor to take the bus-error exception (if the data is required to continue program execution) as described in Section 13.3.12. Note that when a parity error is seen on an AXI control signal, it may or may not be possible to terminate that AXI transaction cleanly, for example, when the parity error is on an ID signal. All parity and uncorrectable errors on the AXI signals are reported on the fault interface as described in Section 29.3.

Correctable single bit errors on read data are corrected on the fly and as the corrected data is sent to the processor, such responses do not carry data bus error. The correctable errors are not signaled on the fault interface, they only set the bit in the internal AXI ECC status register described in Section 29.4.

28.7 AXI Master Atomic Accesses

Atomic accesses involve an atomic update of a memory location for use in inter-process communication. There are two distinct ways to initiate atomic access from the Xtensa processors, with the conditional store synchronization instruction or with the exclusive store option.

28.7.1 Atomic Access with Conditional Store Synchronization Instruction

If the conditional store synchronization instruction option is selected, the Xtensa processor issues atomic operations when a Store Compare Conditional (S32C1I) instruction is executed. Note that the S32C1I instruction behaves differently with AXI3 and AXI4 bridges.

S32C1I with the AXI3 Master

The AXI3 master uses locked transactions to implement S32C1I. It issues a locked read, which is followed by a write that unlocks the bus. The write request is issued with all strobes disabled if the read response data does not match comparison data in the SCOMPARE1 register. Note that the cache attributes for both read and write requests are always set to non-cachable and non-bufferable.

S32C1I with the AXI4 Master

The AXI4 standard does not allow locked transactions, thus the AXI4 master must use a different implementation for atomic accesses using exclusive read and write requests. The following steps describe the implementation:

1. The AXI4 master issues an exclusive read indicated by driving the ARLOCK signal high.
2. If the AXI4 master gets an OKAY (RRESP = 2'b00) response instead of an EXOKAY(RRESP = 2'b01) response, it indicates that the slave does not support exclusive operations. In this case, an address error is sent back to the processor.
3. If the AXI4 master gets an EXOKAY response and the read data does not match comparison data in the SCOMPARE1 register, the AXI4 master returns the value read from memory back to the processor and the transactions ends. The subsequent exclusive store does not happen in this case.
4. If the read data matches the RCW compare data, the AXI4 master does an exclusive store, indicated by driving the AWLOCK signal high.
5. If the write response is OKAY (BRESP = 2'b00), indicating exclusive store failure, then the AXI4 master returns the complement of the compare value (~SCOMPARE1) back to the processor, indicating failure of atomic update.
6. If the response is EXOKAY (BRESP = 2'b01), then the compare value, SCOMPARE1, is returned indicating a successful atomic update.

Note that the AXI4 master always uses normal non-cacheable, non-bufferable cache attributes while doing atomic accesses. If ACE-Lite is configured, the atomic access results in ReadNoSnoop followed by optional WriteNoSnoop, based on the comparison success.

28.7.2 Atomic Access with the Exclusive Store Option

The Exclusive Store option is only supported by the AXI4 bridge. It adds exclusive load (L32EX) and exclusive store (S32EX) instructions to the ISA. Executing an L32EX instruction causes the AXI4 master to issue an exclusive read transaction. Executing an S32EX instruction causes the AXI4 master to issue an exclusive write transaction. These exclusive transactions always have a burst length of 1 and size of 4 bytes. A nor-

mal non-cacheable, non-bufferable cache attribute is used with atomic accesses. If ACE-Lite is configured, L32EX results in ReadNoSnoop and S32EX results in WriteNoSnoop.

Note: The iDMA AXI Master port never issues any atomic transactions.

28.8 AXI Parity/ECC Protection

To protect the interconnect from transmission faults, an additional error-protection layer can be configured to add protection to the AXI bus. The error-protection signals are added to all configured AXI ports; the Main AXI port, the iDMA AXI Master port and the AXI slave port. The protection bits are generated at one end of the bus and checked at the other end of the bus.

ECC code is used to protect RDATA and WDATA signals on the AXI Master and Slave ports. Each 32 bits of data are protected with a 7-bit ECC code.The ECC code generation and checking logic uses the (39,32) Parity-check matrix from the paper titled "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes" by M. Y. Hsiao.

All other control signals are protected using parity codes. One parity bit protects up to eight bits in control signal. For example, Four parity bits are used to protect ARADDR[31:0], whereas one parity bit is used for RVALID.

28.8.1 Choosing Odd/Even Parity

A primary input signal, AXIPARITYSEL, is added to the processor that selects an even or odd parity setting for AXI. When AXIPARITYSEL is 0, even parity is used, when AXIPARITYSEL is 1, odd parity is used. AXIPARITYSEL is latched only during reset. AXIPARITYSEL input must be stable for 10 cycles before and 10 cycles following the de-assertion of reset If the AXI bus interface is configured to use an asynchronous clock, this 10 cycle requirement refers to the slower of the two relevant clocks namely processor clock and bus clock.

AXIPARITYSEL sets the parity setting for all AXI ports configured in the design, including the Main AXI port, the iDMA AXI Master port and the AXI slave port

Note: All handshake signals on all channels, the VALIDs and the READYs, always use odd parity regardless of the AXIPARITYSEL value.

28.8.2 Enabling AXI Parity/ECC Protection

By default, the AXI Parity/ECC protection is off. It can be turned on by writing to the AXI ECC Enable register using the WER instruction. The AXI ECC Enable register and other fault related control registers are described in Section 29.4. Note that ECC enable should only be toggled when there are no AXI transactions in progress.

28.9 AXI Master Example Timing

This section describes the timing of requests issued by the bridge as an AXI master.

28.9.1 Read Requests

Read requests are issued for both instructions and data by the processor. In addition, a request can be for a cache line, or a single word of data. The processor may also try to prefetch data depending on the configuration. The bridge does not encode this information in the AXI read request.

Single-Data Read Request

Figure shows a simple AXI read request with a burst length of 1. This figure shows a load request moving down the processor pipeline and the effect of the AXI access upon the latency of the load request. This example shows an ideal case of a 7-cycles latency for a data request.

- In cycle 5, the load instruction reaches the commit point, but does not commit because it requires data from main memory
- In cycle 6, a PIF read request is issued
- In cycle 7, the bridge issues a read request on the AXI
- In cycle 8, a response is received for the request issued in cycle 7. In most systems, there will be a longer latency to receiving a response, which must be added to the total latency.
- Also in cycle 8, the AXI response is translated directly into a PIF response. The AXI and PIF protocols are fairly compatible, and have similar “valid” and “ready” handshakes, making a direct connection between AXI and PIF possible. Some timing delay is incurred for ID translation.
- In cycle 9, the PIF response is available for the processor (the PIF offers a registered interface for easy system integration). The processor pipeline is able to continue executing with this data.
- In cycle 12, the load instruction completes

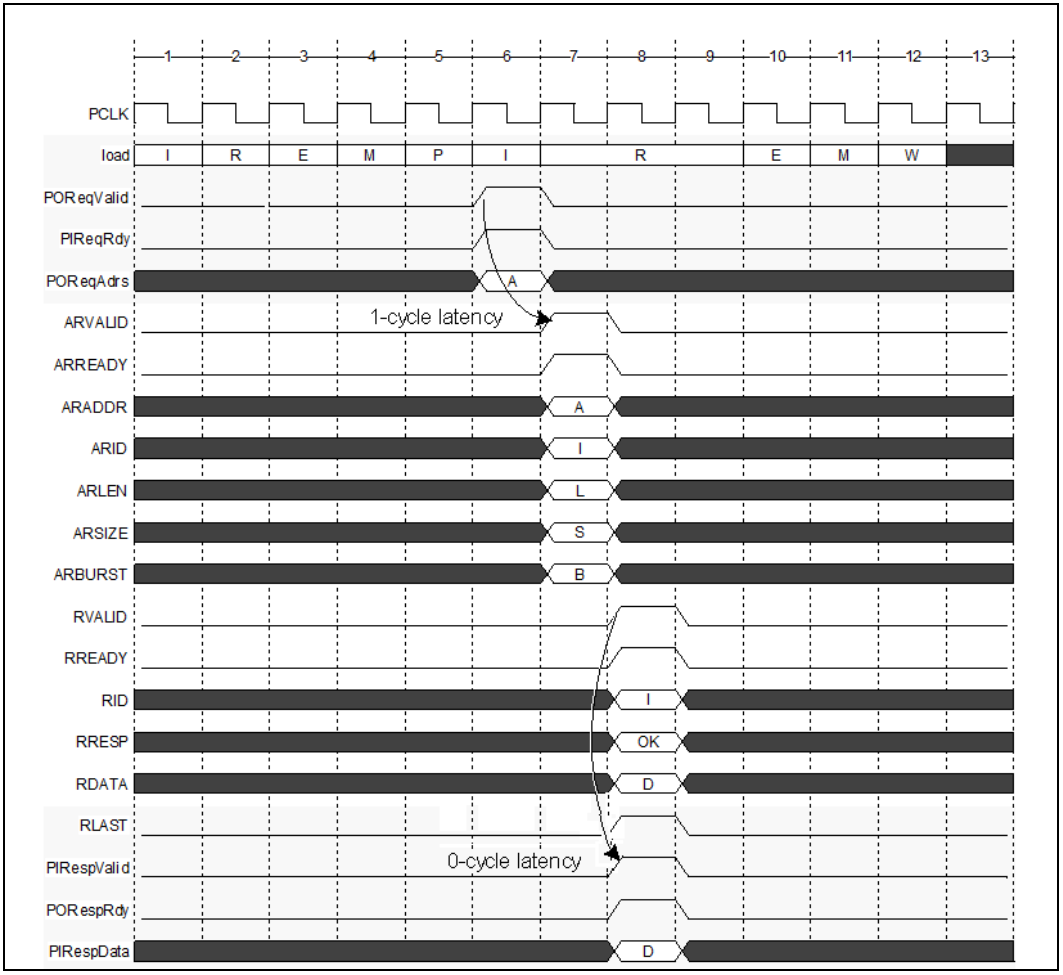


Figure 28–161. AXI Master Single Data Read Request

Burst-Read Request

Longer bursts are issued for cache line requests, or if the bus width is configured to be narrower than the processors cache or local memory data bus widths. For example, consider a processor with a 64-bit instruction width and 64-byte cache line size that is configured for a system bus width of 32 bits. This would result in a burst of two for an uncached instruction fetch, and a burst of sixteen for a cached instruction fetch. If the processor is configured with prefetch, it may issue burst read requests to fetch the data into prefetch memory. Figure 28–162 shows the timing of a burst of four. This example is similar to the example in Figure 28–162.

- In cycle 5, the load instruction reaches the commit point, but does not commit because it misses in the cache and requires data from main memory
- In cycle 6, a PIF read request is issued
- In cycle 7, the bridge issues a read request on AXI
- In cycles 8 through 11, the AXI read responses of four transfers is received. Note that the example shows zero memory latency (ARVALID->RVALID), in most systems, the actual system latency will be longer. These responses are transferred directly to the PIF.
- In cycle 12, the last transfer of the cache line is written to the cache and the load can proceed
- In cycle 15, the load commits with a latency of 10 cycles for a 4-word line

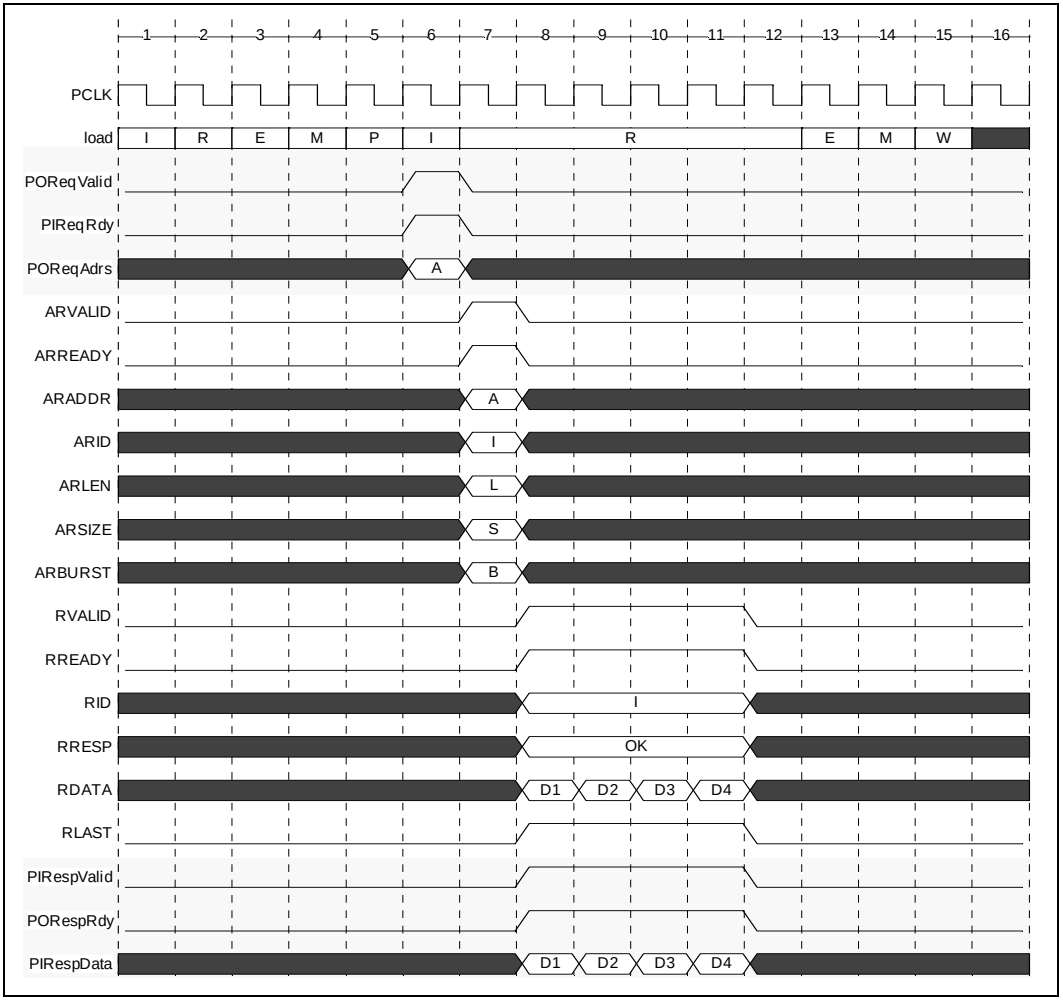


Figure 28–162. AXI Master Block-Read Request

28.9.2 Write Requests

The processor issues data-write requests as a result of a store operation or when a dirty cache line is evicted from a writeback cache. This section describes the timing of writes on AXI.

Back-to-Back Write Requests

Figure 28–163 shows two write requests, the first being a 4-word block write for a dirty-line castout, and the second being a single data write.

- In cycles 2 through 5, the block write request is issued on the PIF. The write data is issued in the same cycles as the address
- In cycle 4, the first transfer of the block write request is issued on the AXI. There is a 2-cycle latency from the PIF request to a write request being issued on the AXI. Note that the address and data channels are simultaneously requested.
- In cycle 5, the data channel request is continuing, but the address channel request completed with a single transfer
- In cycle 6, the second write request is issued on the PIF, a single data request
- In cycle 8, the second write request is issued immediately after the previous write request completed. Note that the address and data channels are always simultaneously requested.
- Also in cycle 8, the response to the block write request is received. Note that the example shows zero memory latency (WVALID->BVALID); in most systems, the actual system latency will be longer. This is the fastest response timing allowed by the AXI protocol, but most systems will have longer response latencies.
- In cycle 9, the response to the second write request is received
- In cycle 10, the first response is sent on the PIF. Read and write responses must arbitrate for the PIF, and the default is that the read response channel is granted the PIF and write responses will have an extra cycle of latency to win arbitration for the PIF
- In cycle 11, the second response is sent on the PIF

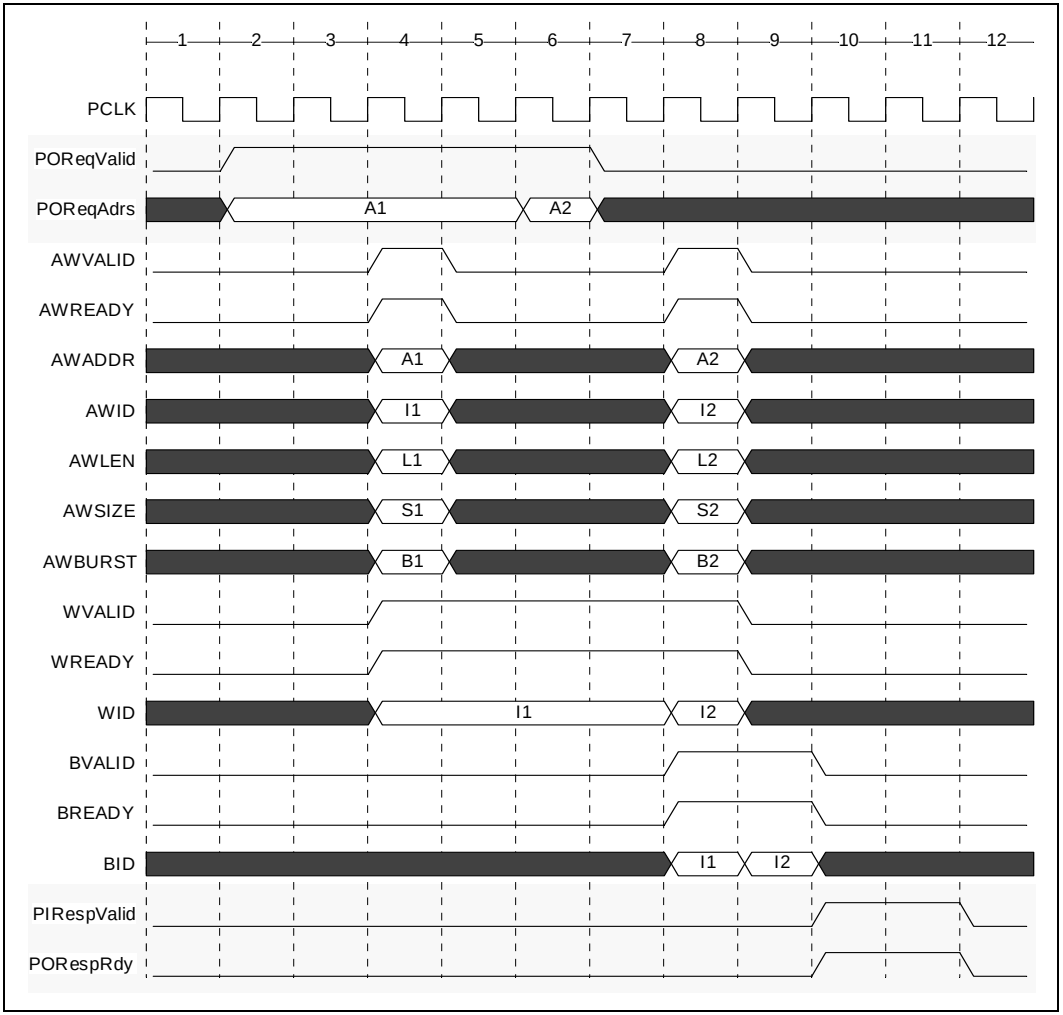


Figure 28–163. AXI Master Write Requests

28.9.3 AXI Parity / ECC Protection

The AXI bus and local memory ECC protection uses the following (39,32) parity check matrix for code generation and checking. The source for this parity-check matrix is from a paper entitled "*A Class of Optimal Minimum Odd-weight-column SEC-DED Codes*" by M. Y. Hsiao."

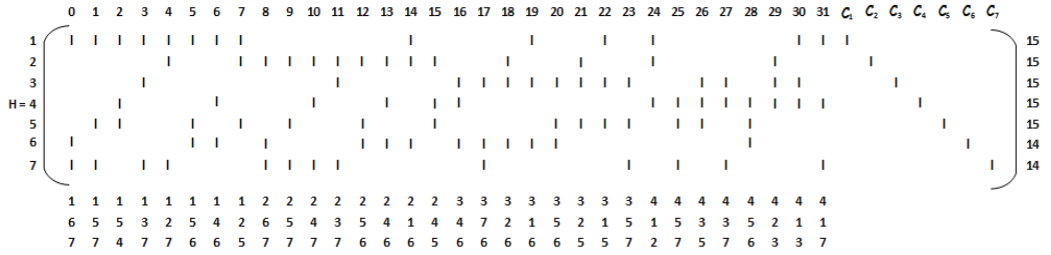


Figure 28–164. Parity Check Matrix

28.10 AXI Slave

The AXI slave port of the bridge is enabled when the processor is configured with a local memory and the Inbound PIF option is selected. The AXI slave port gives other AXI master devices direct access to the local memories of the processor, providing a convenient path to initialize the local memories. In addition, this direct access to the local memories enables novel memory architectures that can operate at much higher performance when cache performance becomes a bottleneck.

The information in this section is generally applicable to all the bridge types, AXI3, AXI4 and ACE-Lite, unless explicitly stated.

28.10.1 AXI Slave Queuing

The AXI slave bridge buffers AXI requests and responses. The depths of these buffers are configurable as described in Section 28.2 “Configuration Options”. The control information from the AXI slave Read Address and Write Address channels is stored in Read Request Control and Write Request Control FIFOs, respectively. The depth of these FIFOs can be set with the AXI Slave Request Control Buffer Depth parameter. The entries in AXI Write Data FIFO can be set with the AXI Slave Request Data Buffer Depth parameter. Finally, the number of entries in AXI read response FIFO can be set using the AXI Slave Response Buffer Depth parameter. Note that the number of entries in the write response FIFO is not a configurable parameter, this FIFO always has 16 entries. The read response FIFO and write response FIFO drive AXI Slave Read Response and Write Response channels respectively.

The buffers described above greatly affect the area and performance of the AXI slave. System designers should configure the FIFO depths based on requirements of a particular system. For example, if the processor is running at a frequency much lower than the bus, having shallow request FIFOs and deep response FIFOs may provide a good area/performance trade off.

28.10.2 AXI Slave Burst Lengths and Alignment

Generally, the AXI slave port provides the best performance if the incoming AXI burst follows this set of rules:

- The burst length must be one of 2, 4, 8, 16. (longer AXI4 bursts are addressed in Section 28.10.3)
- The burst type may be INCR or WRAP, but the address must be aligned by the total number of bytes in the burst
- The burst size must be the entire data bus width

AXI requests that do not follow these rules are split into multiple single transfers, which may perform slower, especially for long burst lengths.

28.10.3 AXI4 Slave Longer Burst Requests

The AXI4 standard widens ARLEN and AWLEN signals to eight bits, allowing requests to have up to 256 transfers. The AXI4 slave bridge supports these longer burst requests, but the best performance is achieved if the following set of rules are followed:

- The AXI4 long burst length must be an integer multiple of 16, excluding 16. For example, one of 32, 48, 64, ..., 224, 240, 256
- The burst type may be INCR or WRAP, but the address must be aligned by the total number of bytes in the burst
- The burst size must be the entire data bus width

The AXI bursts that do not follow these rules may perform slower.

28.10.4 AXI4 Slave Quality of Service and Request priority

The AXI4 slave supports use of Quality of Service signals ARQOS and AWQOS to control the amount of data RAM bandwidth allocated to inbound AXI transaction.
Table 28–114 summarizes the bandwidth allotted to inbound AXI requests to data RAM as a function of the ARQOS and AWQOS signals, assuming that the data RAM width matches the AXI data width. Note that the actual percentage of RAM bandwidth achieved is impacted by static and dynamic factors, such as the RAM-to-AXI width ratio, or how fast the AXI responses can be dispatched, etc. The upper bound of the achievable data RAM bandwidth is the lower of the AXI and data RAM bandwidth

Note: Use a high-priority QoS setting (ARQOS/AWQOS in the range 4'b1100 - 4'b1111) with caution as issuing a long sequence of back-to-back high-priority inbound AXI accesses may stall the Xtensa processor if they both compete for the same data RAM bank.

Table 28–114. AXI4 Slave: Quality of Service

AXI QOS Signals AWQOS[3:0], ARQOS[3:0]	% of Data RAM Bandwidth Allocated to Inbound PIF
4'b0000 - 4'b0011	Approximately 25%
4'b0100 - 4'b0111	Approximately 25%
4'b1000 - 4'b1011	Approximately 25%
4'b1100 - 4'b1111	100%

Inbound AXI requests can proceed in parallel with processor initiated requests if they address separate memories. When both Inbound AXI request and the processor's instruction-fetch operation try to access the same instruction RAM simultaneously, inbound AXI request has higher priority.

28.10.5 AXI3 Slave Request Priority

The AXI3 standard does not support use of Quality of Service signals. Hence, the AXI3 slave issues all local memory accesses with the highest possible priority. This allows the inbound requests to proceed in parallel with processor initiated requests if the requests are addressed to separate memories. Note, however that issuing a long sequence of back-to-back high-priority inbound PIF accesses may stall the Xtensa processor if they both compete for the same data RAM bank.

28.10.6 AXI Slave Request IDs

The AXI slave bridge contains an 16-bit ID field for the read and write channels. The AXI ID width is nominally four bits; additional bits are added as required by interconnect routing elements to encode a proper response route. Small systems may require fewer than 16 bits of ID, in which case unused bits of the ID should be tied off to a constant value.

28.10.7 AXI Slave Locked Accesses

The AXI slave port does not support locked requests, and handles all locked requests as if they were not locked.

28.10.8 AXI Slave Exclusive Accesses

If the Exclusive Store option is selected, the AXI slave port supports incoming exclusive access requests. However, there are certain restrictions on exclusive access support for-example, exclusive access to an IRAM is not supported. If these restrictions are violated, the slave responds to an exclusive load with an OKAY response instead of an EX-

OKAY response. Otherwise, an EXOKAY response is sent and an exclusive access monitor is allocated inside the processor. The exclusive store that follows may succeed and get an EXOKAY response if the monitor indicates that atomicity of the exclusive read-write sequence was maintained.

More details on the exclusive access implementation can be found in Section 4.7.

If the Exclusive Store option is not selected, all incoming exclusive access requests receive an OKAY response instead of an EXOKAY response.

28.10.9 AXI Slave Security Feature

If the AXI Security feature is configured, a primary input, AXISECSLV, is added to the Xtensa processor. This port controls how AXI slave port responds to incoming non-secure requests. If AXISECSLV is asserted and the incoming AXI request is non-secure (A[R/W]PROT[1]=1'b1), the AXI slave port stops it from going to the processor's local memories. A SLVERR response is sent back to such a non-secure transaction. If AXISECSLV is not asserted, non-secure transactions are allowed to access the processor's local memories and they proceed as usual. Secure transactions (A[R/W]PROT[1]=1'b1) are always allowed access to the local memories. Note that AXISECSLV should only to toggled when there are no pending/outstanding transactions buffered in the processor. One way to ensure this is to execute the WAITI instruction and wait until the primary output signal PWaitMode is asserted.

If the AXI Security feature is not configured, the AXI slave port ignores the ARPROT and AWPROT signals.

28.10.10AXI Slave Parity/ECC Protection

To protect the interconnect from transmission faults, an additional error-protection layer can be configured to add protection to the AXI bus. The implementation for the AXI slave port is very similar to that in the AXI master port as described in Section 28.8. Parity bits are used to protect control signals and ECC code is used to protect data. Valid and Ready signals always use odd parity, but the parity type for other control signals can be selected using the AXIPARITYSEL signal described in Section 28.8.1.

28.10.11AXI Slave Response Errors

The AXI slave port reports errors on requests with a response of SLVERR. Errors on write requests can only be reported if the Inbound PIF Write Response option is selected. Errors can either be address errors for requests outside of a local memories address space, or data integrity errors detected when ECC or parity is configured. In addition, requests to instruction RAMs must be multiples of 32 bits.

If the AXI Parity/ECC protection is configured, the AXI slave port checks for transmission errors on incoming control and data signal. If a parity error is observed on any of the required control signals, or if an uncorrectable ECC error is found in the data, the bridge responds with a SLVERR response. All parity and uncorrectable errors on the AXI signals are reported on the fault interface as described in Section 29.3.

Correctable single bit errors on the write data are corrected on the fly and since the corrected data is written to the local memories of the processor such responses do not carry SLVERR. The correctable errors are not signaled on the fault interface, they only set the bit in the internal AXI ECC status register described in Section 29.4.

If the AXI Security feature is configured, all non-secure requests (ARPROT[1]=1'b1 and AWPROT[1]=1'b1) receive an SLVERR response when AXISECSLV input is asserted.

28.10.12AXI Slave Additional Control Interfaces

The AXI slave port does not implement the cache support interface and ignores incoming ARCACHE and AWCACHE signals. The AXI slave port also ignores the privilege (A[R/W]PROT[0]) and instruction (A[R/W]PROT[2]) bits in the protection interface.

If the AXI Security feature is configured, the AXI slave port uses the secure bit (A[R/W]PROT[1]) in the protection interface as described in Section 28.10.9. Otherwise, the secure bit is ignored.

28.10.13AXI Slave Idle Mode

The AXI slave port implements an idle mode as a low-power feature. The bridge enters the idle mode if it does not see a valid AXI request on either the read channel or the write channel for 1024 cycles. During the idle mode the slave bridge de-asserts its ready signals, ARREADY and AWREADY. The bridge comes out of the idle mode when an AXI request is seen on either of the channels. The very first AXI request after the wake up incurs a two cycle delay because of the idle mode.

28.11 AXI Slave Timing Diagrams

Generally, the AXI slave port provides the best performance if the incoming AXI burst follows the set of rules in Section 28.10.2 and maps the AXI burst requests into PIF block requests.

28.11.1 AXI Read Request that Maps to PIF Block Read Request

Figure 28–165 shows example timing for an AXI3 read burst that maps to a PIF block read request.

- In cycle 2, the AXI burst read request is received. The length of the AXI burst is four transfers; otherwise it meets the size and address alignment requirements to map to a PIF block read request.
- In cycle 3, the PIF block read for four transfers is issued. There is a 1-cycle latency to issue the PIF request
- In cycles 5 through 8, the local memory is accessed. Data is returned one cycle after each access. There is a 2-cycle latency from the PIF request to the first local memory access
- In cycles 7 through 10, four PIF response transfers are issued. There is a 2-cycle latency from the local memory access to a PIF response for a 5-stage pipeline, and a 3-cycle latency for a 7-stage pipeline.
- In cycles 8 through 11, four AXI response transfers are issued. There is a 1-cycle latency from the PIF response to the AXI response.

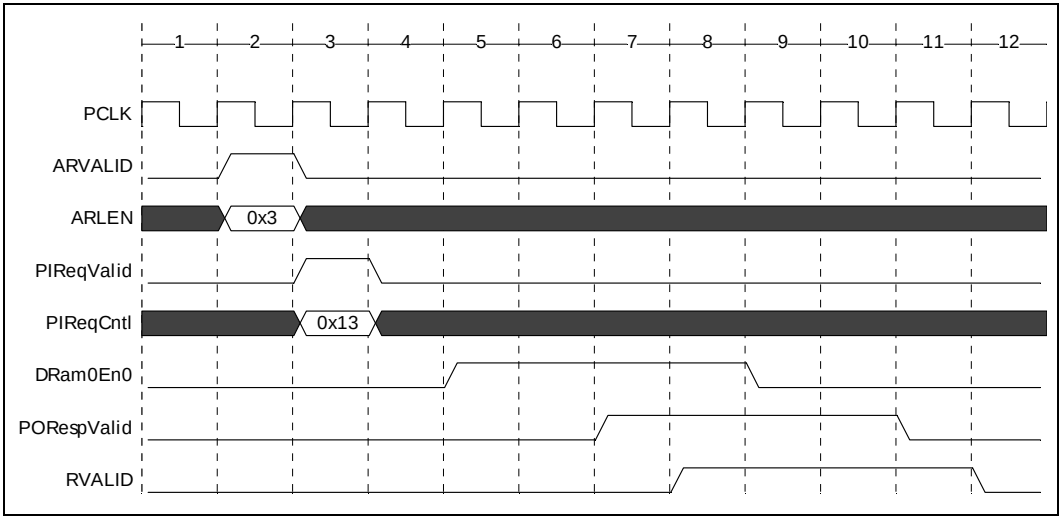


Figure 28–165. AXI3 Slave Read Request that Maps to PIF Block Read

28.11.2 AXI Write Request that Maps to PIF Block Write Request

Figure 28–166 shows example timing for an AXI write burst that maps to a PIF block write request. Note that the inbound PIF port of the processor always supports Arbitrary Byte Enables, thus the AXI strobe signal can be directly passed on.

- In cycle 2, the AXI write request is received. In this example, the address and data channels are simultaneously requested, however this is not a requirement of the bridge or the AXI protocol. The length of the AXI write burst is two transfers; otherwise it meets the size and address alignment requirements to map to a PIF block read request.

- In cycles 2 and 3, the two words of data are received, note that the write strobes are allowed to sparse, inbound PIF port supports them
- In cycle 3, the PIF block write for two transfers is issued. There is minimum1-cycle latency to issue the PIF request, although since AXI Read and Write requests arbitrate for single PIF request channel, the latency may be more.
- In cycle 4, the last word of the PIF block write request is driven on PIF
- In cycles 5 and 6, the local memory is written
- In cycle 7, a PIF write response is issued
- In cycle 8, an AXI write response is issued

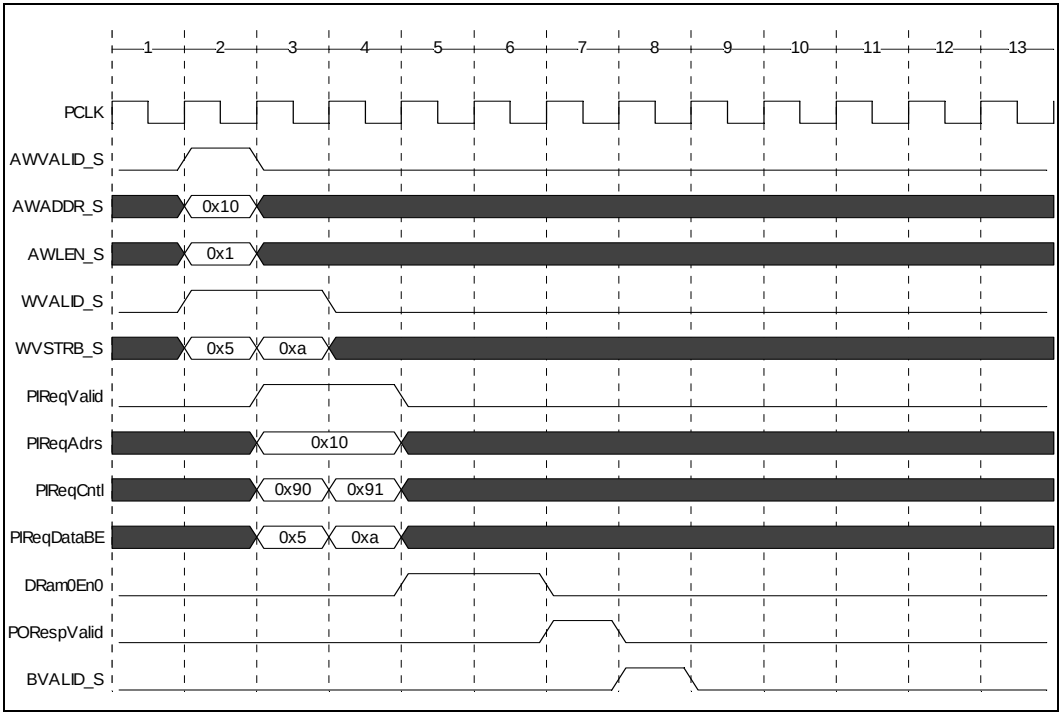


Figure 28–166. AXI3 Slave Write Request that Maps to PIF Block Write

28.11.3 AXI Bursts Broken Into Multiple Single Requests

If the incoming AXI burst does not follow the set of rules explained in Section 28.10.2, the burst is broken into back-to-back multiple single PIF requests.

Figure 28–167 shows a read request that does map to a PIF block read request.

- In cycle 2, an AXI read request is received with a length of 3. Only lengths of 2, 4, 8, or 16 map to a PIF block read request.

- In cycle 3, the first PIF single-data read request is issued, with ID 0x20
- In cycle 4, the second PIF single-data read request is issued, with ID 0x21
- In cycle 5, the third PIF single-data read request is issued, with ID 0x22
- Also in cycle 5, the first request to local memory is issued
- In cycle 6, the second request to local memory is issued
- In cycle 7, the third request to local memory is issued
- Also in cycle 7, the first PIF response is issued
- In cycle 8, second PIF responses are issued
- Also in cycle 8, the first AXI response is issued
- In cycle 9, last PIF responses are issued
- Also in cycle 9, the second AXI response is issued
- In cycle 10, the last AXI read response is issued

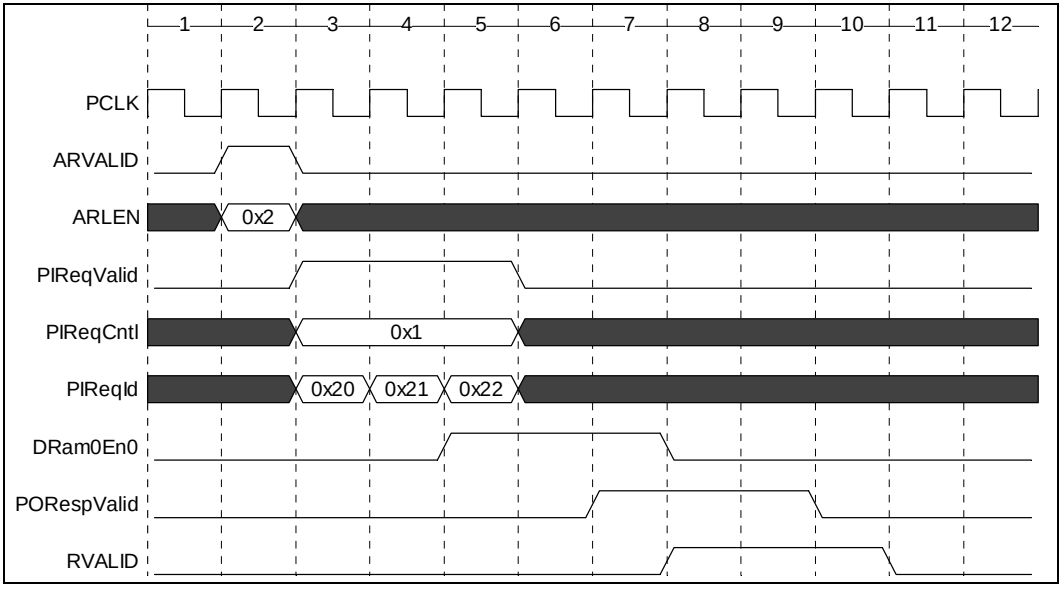


Figure 28–167. AXI3 Slave Read Multiple Requests that Maps to PIF Block Read

28.11.4 AXI3 Write Request that does not Map to a PIF Block Write Request

Figure 28–168 shows a 3-word burst write request.

- In cycle 2, the AXI write request is received with a length of 3. Only lengths of 2, 4, 8, or 16 may map to a PIF block write request.
- In cycles 2 through 4, the three words of data are received

- In cycle 3, the first PIF write request is issued. The address matches the beginning address of the AXI burst request. There is a minimum 1-cycle latency from the request on the AXI write data channel to the request appearing on the PIF. Note that the byte enables are derived from AXI strobes.
- In cycle 4, the second PIF write request is issued, and the address is incremented by four bytes. The processor can accept multiple outstanding requests, and will pipeline the requests.
- In cycle 5, the last PIF write request is issued
- From cycles 5 to 7, three requests are issued to the local memory
- From cycles 7 to 9, three PIF responses are issued
- In cycle 7, the second request is issued to local memory, and the first PIF response is issued
- In cycle 10, the AXI write response is issued

.

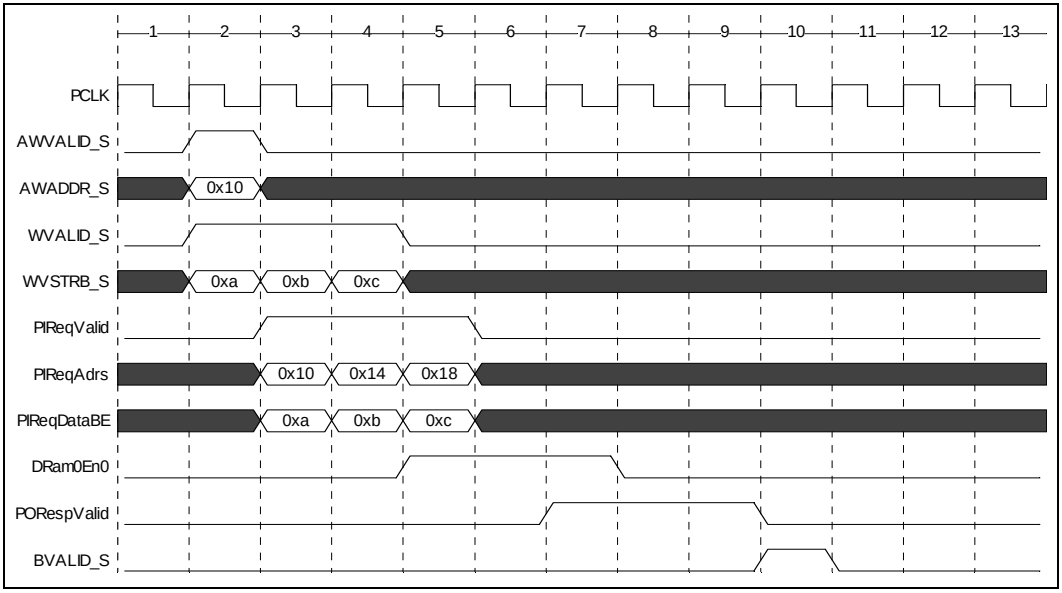


Figure 28–168. AXI3 Slave Write Request that does Not Map to PIF Block Write

29. Fault Handling and Error Reporting

The Xtensa processor is designed to detect and report some fault conditions, including a double exception fault or a Parity/ECC error on one of the AXI ports. Note that the AXI Parity/ECC feature must be configured to detect and report transmission errors on the AXI bus.

29.1 Fatal and Non-fatal Faults

Faults detected by the processor fall into the fatalor non-fatal category as follows:

- **Fatal faults**: These are the type of faults that the processor can detect, but cannot recover from without the system's help. In these situations, the processor does not trust itself to continue executing reliably, even in an exception handler. These errors may be internal to the processor or caused by the system. An example of the former, is taking a second uncorrectable ECC error while servicing the first uncorrectable ECC error inside the exception handler; an example of the latter is detecting a parity error on the RVALID field ofthe AXI Master port. When a fatal error occurs, the Xtensa processor uses primary output signals to notify the system. More information about the fault is also provided through other output signals. The same information is also logged in an Xtensa processor's internal fault information register.

  Note: The Xtensa processor does not take any exception/interrupt when fatal faults happen (unless the only fatal fault is a double exception). Instead, it notifies the system that a fatal fault has occurred and that the system needs to take corrective action.

- **Non-fatal faults**: These are the type of faults that the processor can detect and can recover from or report its condition to the system using a software mechanism. In this case, the processor can safely continue execution. The reporting happens using the following mechanisms:

  a. The processor takes an exception or interrupt and can communicate the error to the system or take self-correction action by running the error handler routine. Most exceptions are precise with the following exceptions:

     - Cache memory errors for a castout line
     - Write response errors (reported by Write Error Interrupt)

  b. The processor asserts an I/O signal: When an error is detected on the prefetch RAM, an I/O signal is pulsed for a single cycle, PrefetchRamErrorUncorrected for a double bit error and PrefetchRamErrorCorrected for a single bit error. In both cases it invalidates the offending prefetched line, discards the data, and continues execution, obtaining the needed data from another source.

29.2 Fault Detection

The fault handling logic can detect two different types of faults. The first is double exception fault. Fault handling logic detects this fault by monitoring the program flow. Every time the processor enters the double exception handler code, this fault is detected.

The other type is AXI transmission fault Parity or ECC checks are used to detect ~~for~~ transmission errors on all AXI input signals. Following are more details on how AXI transmission faults are detected:

- Parity protection signals for incoming Valid signals are checked every cycle.
- Parity protection signals for incoming Ready signals are checked when the corresponding Valid signal is asserted.
- Parity/ECC protection signals for other control and data signals are registered during the request/response cycle and checked.
- Read data received by the AXI master and write data received by the AXI slave are checked using ECC code.
- Single bit errors are corrected on the fly and double bit errors are detected.
- All other control signals (except for Valid and Ready) are also checked using parity.

29.3 Fault Reporting

The Xtensa processor monitors the fault condition on the AXI bus interface and inside the core. When a fault occurs, the information is aggregated into a 32-bit encoding, which is placed in the internal Fault Information Register (FIR). The same 32-bit encoding is also driven on a primary output signal called PFaultInfo. Based on the severity of the fault, the processor may also decide to send a fatal error notification to the system. For all faults that need immediate system attention,the Xtensa processor asserts a primary output signal called PFatalError. Other fault scenarios that can be dealt with internally are silently recorded in the (FIR), PFatalError is not asserted in those cases. Table 29–115 summaries the signals used for fault reporting.

Table 29–115. Fault Reporting Signals

Signal Name	Width	Direction	Description
PFatalError	1	Output	Sticky fatal error notification signal that is asserted when a fatal error condition occurs (e.g., parity error on AXI handshake signal, or core trapped in the double exception vector). It is expected that the system will use the information presented on the PFaultInfo signal and decide whether to reset the core or the whole system. The PFatalError signal is cleared by core reset.
PFaultInfo	32	Output	Fault information signal. This signal mirrors the processor's internal FIR, using the same bit encoding to provide the source and severity of the fault. See Section 29.3.1 for details. This signal is set to the encoding of the most recentand most severe fault. The PFaultInfo signal is cleared by core reset.
PFaultInfoValid	1	Output	Strobe signal that is asserted for one cycle every time the PFaultInfo signal changes its value. The system can use it as an enable to capture PFaultInfo.
DoubleExceptionError	1	Output	Single cycle assertion for every time a double exception fault occurs. Unlike PFatalError, this signal is not sticky

29.3.1 The PFaultInfo and Fault Information Register

The Fault Information Register (FIR) provides more information on the error condition and the PFaultInfo output signal mirrors this internal register. FIR can be read by the Xtensa processor through internal an ERI interface. Some bits of FIR can also be written by the Xtensa software. The system cannot access Xtensa's FIR, but can sample it externally each time it is updated with a new value as indicated by the PFaultInfoValid strobe signal. Table 29–116 shows the current set of PFaultInfo / Fault Information Register encodings. Note that more faults may be added to this table later.

Table 29–116. PFaultInfo / FIR Encodings

PFaultInfo / FIR Encoding					PFatalError		
Bits 31:21*	Bit 20*	Bits 19:16	Bits 15:8*	Bits 7:0*	Fault	Asserted?	Description
11'd0	1'b0	UUUU	0000_0000	0000_0000	No Fault	No	No fault has occurred
11'd0	M	UUUU	0010_SSSS	0000_0000	Double Exception	Yes	Theprocessor is executing code from the double exception vector. The fatal error notification (PFatalError) is asserted requesting assistance from the System. The severity level is set to core reset, 0001. Optionally, software can write to the user bits and communicate with the system to abort core reset.
11'd0	M	UUUU	0001_SSSS	CCCC_PPPP	Transmission Fault on AXI	▪ Yes: If SSSS indicates severity as System/Core Reset ▪ No: If SSSS indicates fault severity as No Reset	A parity error or an uncorrectable ECC Error has happened on one of the AXI ports. The encodings provide information about the AXI port and the AXI channel within the port that had the fault. A severity assessment is also provided.
*Read-Only for Xtensa SW							

The terminology used in the bit encodings are explained in Table 29–117.

Table 29–117. Descriptions for PFaultInfo Bits

Bits	Meaning	Description
SSSS	Fault Severity Level	▪ 0000: Fault Severity is not reset. The system can recover without resetting. ▪ 0001: Fault Severity is core reset. Minimally,reset the Xtensa processor to recover from the fault. ▪ 0010: Fault Severity is system reset. The whole system may need to be reset to recover from the fault. Note: The suggested severity level and the recovery mechanism are recommendations and may not always work.
CCCC	AXI Channel	▪ 0000: Read address AXI channel ▪ 0001: Write address AXI channel ▪ 0010: Write data AXI channel ▪ 0011: Read data AXI channel ▪ 0100: Write response AXI channel
PPPP	AXI Port	▪ 0000: Main AXI master port ▪ 0001: Secondary (iDMA) AXI master port ▪ 0010: AXI slave port
UUUU	User bits	Software running on Xtensa can write to the user bits. These bits are zero by default.
M	Multiple Faults	▪ 0: Only one fault has occurred so far. ▪ 1: Multiple faults have occurred. In this case, the FIR register has information only on the most severe fault.

29.3.2 User Bits in the FIR/PFaultInfo Signal

The user-bit field (bits [20:16]) in the FIR and PFaultInfo signal is writable by the software running on the Xtensa processor. This field is useful in cases when a fatal fault occurs, but somehow software running on the processor is still in control. Software can write to the user bits and communicate with the system. This is mostly relevant for double exception faults, however not for any of the bus Parity / ECC faults.

Note that not all double exceptions are fatal. Sometimes the processor may encounter a double exception, execute the code in the double exception handler and recover. The system should only reset the processor if the double exception is truly unrecoverable. Employ the user bits in the FIR to distinguish between these two cases.

For example, a double exception occurs, PFatalError is asserted and PFaultInfo suggests processor reset, the user bits in PFaultInfo are initially zero. If the processor is not truly dead and the software is in control, the double exception handler code can immediately

set one or more of the user bits in the FIR. The external fault processor can monitor at the user-bit field on PFaultInfo and decide to abort or proceed with an Xtensa core reset based on their status.

The DoubleExceptionError signal described in Table 29–115 is also useful in dealing with double exception faults. This non-sticky signal provides a single-cycle pulse every time a double exception occurs. If frequent double exceptions keep occurring, the system can decide to reset the processor regardless of the status of the user bits.

29.3.3 Dealing with Multiple Faults

When multiple faults occur in quick succession before the system can intervene, they are prioritized according to the severity. Forexample, a fault that may requirea complete system reset is preserved over a fault that can be recovered by just resetting the Xtensa processor, regardless of the order in which they occur.

If multiple faults occur, a bit is also set in FIR (bit 20) as described in Section 29.3.1 “The PFaultInfo and Fault Information Register”.

29.4 Other Internal Registers

Apart from FIR, there are other internal registers that control Parity/ECC related functionality, as listed in Table 29–118.Note that only the Xtensa processor can access these registers using it’s ERI interface and that the external masters do not have access to these registers.

Table 29–118. Internal Fault Status/Control Registers

Register	ERI Address	Access	Reset Value	Description
Fault Information Register	0x00103030	R (all bits) W (only bits 19:16)	0x0	See Section 29.3.1 for details.
AXI ECC Enable	0x00103034	R/W	0x0	▪ Writing 1 enables AXI Parity/ECC checking ▪ Writing 0 disables AXI Parity/ECC checking. ▪ The register affects all AXI ports. When AXI Parity/ECC checking is disabled, parity and ECC code checking on the inputs does not happen, however Xtensa continues to generate parity and ECC codes for its outputs. Note: ECC enable should only be toggled when there are no AXI transactions in progress
AXI ECC Status	0x00103038	R	0x0	▪ Bit 0 is set to indicate that an uncorrectable error has occurred ▪ Bit 1 is set to indicatethat a correctable error has occurred

30. Xtensa Software Tools Overview

The Xtensa Processor Generator (XPG) creates processor RTL, test benches, and monitors for hardware simulation. However, the Xtensa processor is more than a hardware design. The XPG also creates a complete software design environment tailored to the configured processor. This environment includes tools for writing applications, for debugging and profiling programs, and for managing processor configurations.
Table 30–119 summarizes the development tools provided with the processor RTL.

The tools include the Xtensa instruction set simulator (ISS) and several processor simulation models. System designers may use these models to simulate their entire system at different levels of precision.

Table 30–119. Summary of Xtensa Development Tools

Tool	Description
For Application development:	
Xplorer version	Xtensa Xplorer Development Environment - Standard Edition
`xt-xcc`	Xtensa C Compiler
`xt-xc++`	Xtensa C++ Compiler
`xt-ld`	GNU Linker
`xt-as`	GNU Assembler
For Application Simulation:	
Xplorer version	Xtensa Xplorer Development Environment - Standard Edition
`xt-run`	Xtensa Instruction Set Simulator
`xt-gdb`	GNU Debugger
`xt-gprof`	GNU Profiler
For Configuration Management:	
Xplorer version	Xtensa Xplorer Development Environment - Processor Developer's Edition
For System Simulation:	
XTMP API	Xtensa Modeling Protocol
XTSC	Xtensa SystemC package
`xtsc-run, xtmp-run`	Pre-built system simulators

30.1 System Simulation Glossary

Note: *The following definitions are generic in nature and are not specific to any* solutions *provided by Cadence.*

- *Hardware (HDL) Simulator:*
  A simulator for a hardware description language. HDL simulation is slow but very accurate.
- *Instruction Set Simulator (ISS):*
  A high-performance, software-only processor model and simulator. An ISS runs programs compiled for the actual processor, executes all instructions, and produces the same results as the processor. ISS models are very fast, but may not produce cycle-count timing results as accurately as HDL simulations. Some ISS models simulate only at the instruction level (they do not model the pipeline) and provide only approximate cycle counts.
- *Processor Co-Simulation Model (CSM):*
  A model of a processor's external interface that performs all of the processor's bus transactions. Some CSMs are simple, script-driven models. Others are controlled by an ISS and can run any processor program.
- *Software Co-Simulation:*
  An environment in which a software model of a processor drives a CSM, which connects to other HDL components in a hardware simulator. Co-simulation can be much faster than simulating the entire system with an HDL simulator. Also known as hardware/software co-verification.
- *Bus transaction-level model:*
  A model of a processor bus in which all operations are represented by high-level transactions, rather than values on bus signals.
- *Bus signal-level model:*
  A model of a processor bus that accurately models all of the bus-interface signals.

30.2 Xtensa System Models

Cadence provides several models for the Xtensa processor that can be used for system simulation. These simulation models range from high-level software simulations of the processor to the synthesizable HDL code that defines the processor. While it is possible to simulate a large multiple-processor system entirely in HDL, the simulation would run very slowly. Developing and debugging software is also harder using hardware simulation. Table 30–120 lists system-modeling alternatives for designs based on Xtensa processors and the relative speeds of these alternatives.

Table 30–120. Instruction Set Simulator Performance Comparisons with Other Tools

Simulation Speed (MIPS or cycles/seconds)	Modeling Tool	Benefits
20 to 50 MIPS[1]	Stand-alone ISS in TurboXim mode	▪ Fast functional simulation for rapid application testing
1.5 to 40 MIPS[1,2]	XTMP or XTSC in TurboXim mode	▪ Fast functional simulation for rapid application testing at the system level
800K to 1,600K cycles/second	Standalone ISS in cycle-accurate mode	▪ Software verification ▪ Cycle accuracy
600K to 1,000K cycles/second[2]	XTMP in cycle-accurate mode	▪ Multi-core subsystem modeling ▪ Cycle accuracy
350K to 600K cycles/second[2]	XTSC in cycle-accurate mode	▪ Multi-core subsystem modeling ▪ SystemC interfaces ▪ Cycle accuracy
1K to 4K cycles/second	RTL Simulation	▪ Functional verification ▪ Pipeline-cycle accuracy ▪ High visibility and accuracy
10 to 100 cycles/second	Gate-level Simulation	▪ Timing verification ▪ Pipeline-cycle accuracy ▪ High visibility and accuracy

1. TurboXim mode simulation speed is an estimate for relatively long-running application programs (1 billion instructions or more).

2. Simulation speed is an estimate for a single Xtensa core in XTMP or XTSC.

Reviewed below are two different system-simulation models for the Xtensa processor. Both are based on the *Xtensa Instruction Set Simulator*. The Xtensa ISS is a fast, cycle-accurate model and processor simulator. By itself, the ISS allows developers to run and debug Xtensa processor programs. Three simulation packages add interfaces and libraries to the ISS to enable a designer to simulate a full system:

- *Xtensa Modeling Protocol (XTMP):*

  A programming interface (API) and library for the Xtensa ISS. XTMP allows developers to write their own C or C++ models of components that connect to Xtensa processors. XTMP provides a very efficient simulation environment, suitable for multiprocessor system simulations.

- *Xtensa SystemC Package (XTSC)*

  The Xtensa SystemC package (XTSC) supports both transaction-level modeling (TLM) and pin-level modeling of Xtensa cores in a system-on-a-chip (SOC) SystemC simulation environment or in a SystemC-Verilog co-simulation environment.

Each Xtensa memory, TIE, and certain system I/O interfaces can be individually selected to be exposed as a TLM interface or as a pin-level interface. The XTSC package includes the following items:

- The XTSC core library and the associated header files needed to use it. Among other things, this library gives access to the Xtensa Instruction Set Simulator (ISS) in the form of a SystemC module and to the TLM interface classes used for transaction-level intermodule communication.
- The XTSC example component library and all source and header files needed to use or re-build it. This library contains a set of configurable SOC example TLM components such as memories, arbiters, routers, queues, lookups, etc. in the form of SystemC modules. It also contains memory, queue, and lookup SystemC modules that have pin-level interfaces. In addition, it contains a set of test bench modules that can drive any XTSC TLM or pin-level interface using script files.
- A set of example projects illustrating each of the Xtensa core and SOC TLM and pin-level component modules in a pure SystemC environment.
- A set of example projects illustrating co-simulation between XTSC models and Verilog models for each of the Xtensa memory, TIE, and certain system I/O interfaces.
- The `xtsc-run` program to quickly build and simulate simple to complex XTSC systems or to generate all the files necessary to co-simulate an XTSC system with one or more existing Verilog models ("SystemC on top") or to co-simulate an existing Verilog system with one or more XTSC models ("Verilog on top").

Note: The XTMP and XTSC APIs and the Xtensa CSM are provided only to Cadence licensees who have purchased the system simulation and multiprocessor support option for the Xtensa processor.

30.3 Cycle Accuracy

The XTMP environment, XTSC package, Pin Level XTSC, and CSM models are all built upon the Xtensa processor ISS. Each Xtensa ISS is a cycle-approximate model and simulator for a configured Xtensa processor. It accurately models the processor pipeline and simulates the interaction of the several instructions that occupy the processor's pipeline at any time. The ISS models caches, pipeline stalls, and resource contention. Used with the XTMP API or the XTSC package, the ISS can interact with cycle-accurate models of memory or other system devices to help predict the performance of an entire system.

30.4 Xtensa System Simulation Support

Cadence provides two application-programming interfaces (APIs) for accessing the Xtensa Instruction Set Simulator (ISS): the Xtensa Modeling Protocol (XTMP) environment and the Xtensa SystemC (XTSC) package. These APIs allow the SOC design team to perform fast and accurate simulation of designs incorporating one or more Xtensa processor cores. The XTMP simulator runs orders of magnitude faster than RTL simulators and is a very powerful system-design and software-development tool. The XTSC package supports both transaction-level modeling (TLM) and pin-level modeling of Xtensa processor cores in a SystemC simulation environment and in a SystemC-Verilog cosimulation environment.

30.4.1 The Xtensa ISS and XTMP

The XTMP environment allows system designers to create customized, multi-threaded simulations to model more complicated systems. SOC designers can instantiate multiple Xtensa processor cores and use XTMP to connect these processor cores to designer-defined peripherals and interconnects. XTMP can be used for simulating homogeneous or heterogeneous multiple-processor subsystems as well as complex uni-processor architectures. Figure 30–169 shows a simple system simulation built with XTMP.

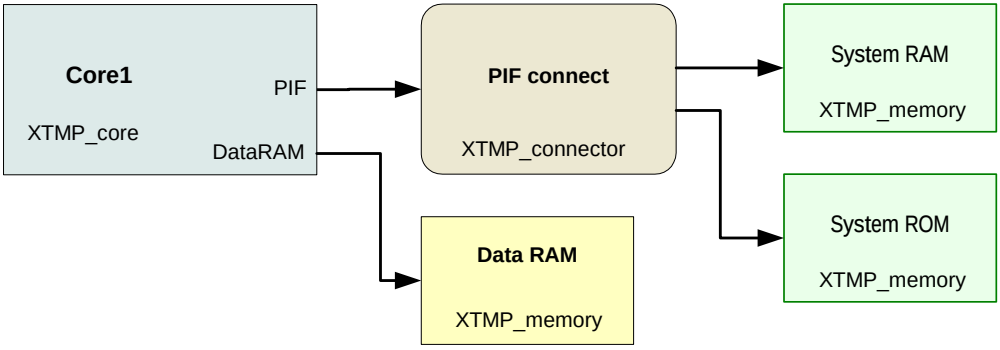


Figure 30–169. A Simple System Built with XTMP

The XTMP environment allows system designers to create, debug, profile, and verify their integrated SOC and software architecture early in the design process instead of waiting until the end of a design project to integrate the hardware and the software.

Because XTMP device models are written in C and C++, the XTMP simulation runs faster than an HDL simulator running RTL models. XTMP device models are also much easier to modify and update than hardware prototypes.

A module written using the XTMP API is a *transaction-level* model. It represents bus and memory accesses using high-level transactions rather than by wiggling bus signals. A designer can write XTMP device and memory modules as cycle-accurate models or as untimed functional blocks.

30.4.2 XTSC, the Xtensa SystemC Support Package

The XTSC package works with the Xtensa ISS and includes the following items:

- The XTSC core library, which gives access to the Xtensa ISS in the form of a SystemC module (`sc_module`).
- The XTSC component library, which contains a set of configurable SOC example components in the form of SystemC modules. Components with transaction-level interfaces include memories, arbiters, routers, queues, lookups, etc. Components with SystemC pin-level interfaces are memories, queues, and lookups.
- The `xtsc-run` program, which can be used for two distinct purposes:
  - To quickly build and simulate arbitrary XTSC systems comprised of any number of modules from the XTSC core or component library, or
  - To generate a glue layer needed for co-simulation of an XTSC system with one or more existing Verilog modules (SystemC-on-top), or co-simulation of an existing Verilog system with one or more XTSC modules (Verilog-on-top).
- A set of example projects that illustrate Xtensa core and component library transaction-level and pin-level interfaces in a pure SystemC environment, and a set of example projects that illustrate co-simulation between XTSC and Verilog modules.

Figure 30–170 shows a transaction-level XTSC model of two Xtensa cores communicating through a FIFO using TIE queues. The producer (xtsc_core core0) writes data to the FIFO (xtsc_queue Q1) using a TIE output queue (OUTQ1), which is modeled by the nb_push() transaction-level interface. The consumer (xtsc_core core1) reads the data using a TIE input queue (INQ1), which is modeled by the nb_pop() transaction-level interface:

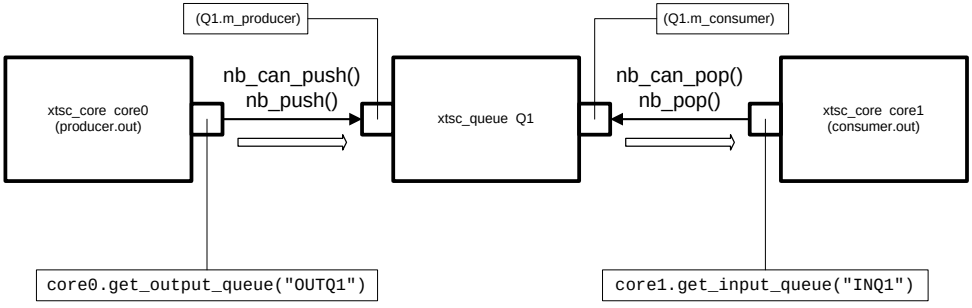


Figure 30–170. Example of xtsc_queue

The same system modeled using XTSC pin-level interfaces is shown in Figure 30–171. TIE output queue OUTQ1 is modeled with SystemC signals, whose names are the same as the corresponding Xtensa processor signals: TIE_OUTQ1_PushReq, TIE_OUTQ1, TIE_OUTQ1_Full; and TIE input queue INQ1 is modeled with TIE_INQ1_Pop_Req, TIE_INQ1, TIE_INQ1_Empty SystemC signals:

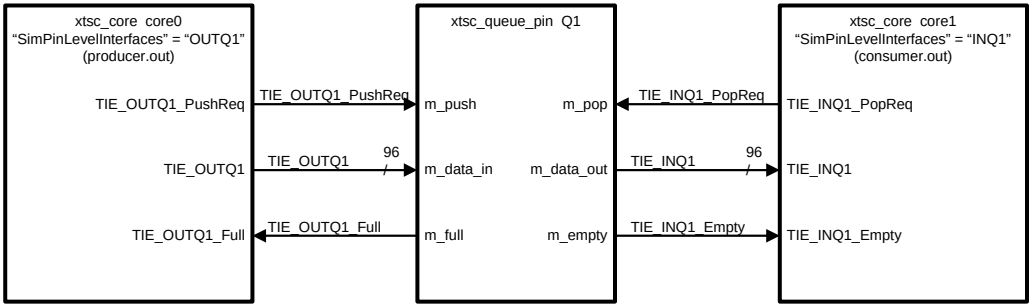


Figure 30–171. Example of xtsc_queue_pin

Figure 30–172 shows a co-simulation example, with two Xtensa core software models connected via XTSC pin-level interfaces and a SystemC proxy to a Verilog implementation of a FIFO (queue.v).

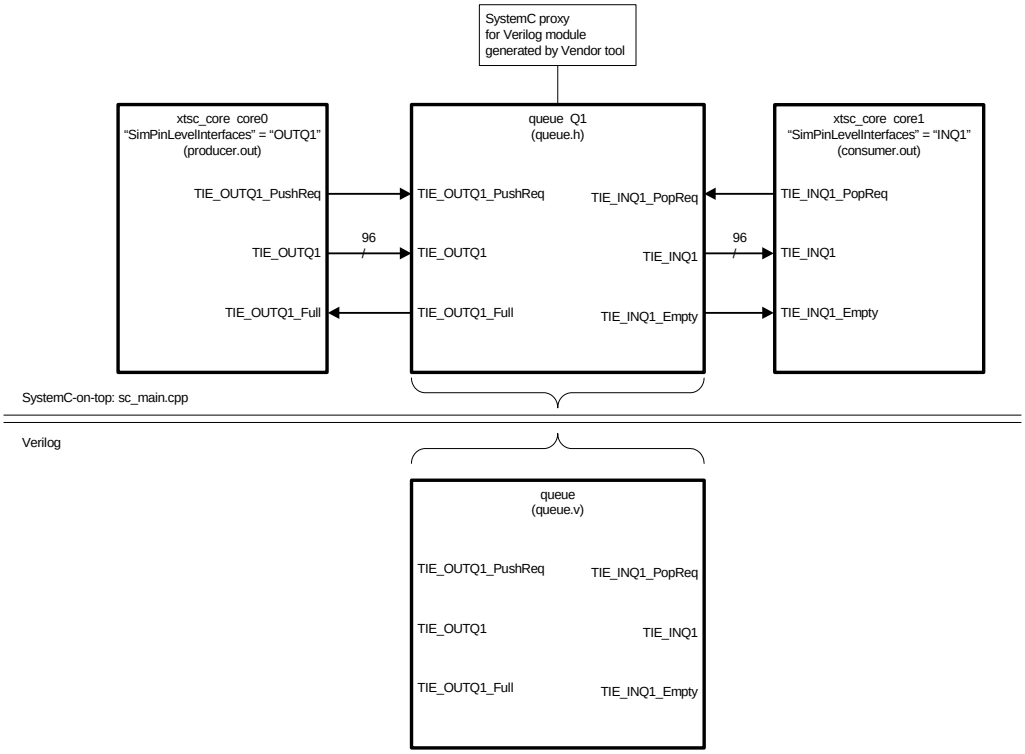


Figure 30–172. Example of xtsc_queue_pin in SystemC-on-top Cosimulation

30.4.3 Xtensa Co-Simulation Model for Mentor Graphics Seamless

The Mentor Seamless Coverification Environment (CVE) is not supported by the full Xtensa LX7 product release. For further information on earlier versions that support this, contact your nearest Cadence Sales office.

31. FPGA Emulation Flow

It is often desirable to emulate a design or to explore the design space using an FPGA prior to taping out a chip. Implementing a design on an FPGA can be a tedious process so Cadence provides an FPGA deliverable designed to help designers quickly integrate Xtensa processors into a target system for a given FPGA family. When FPGA emulation support is selected in Xplorer, Cadence provides a synthesized netlist of the Xtensa processor for the FPGA architecture selected at license time along with all of the necessary RTL files and scripts to create a simple self-contained processor system for the FPGA. Because this packaging is not tied to any specific board design, designers can choose to use the provided sample system as a starting point or to freely instantiate Xtensa configurations in emulation systems that utilize Xilinx FPGA. An optional bit stream for emulation boards, such as KC705, is also available.

The provided sample system in Figure 31–173 is designed to be used within a single FPGA and can be quickly implemented using the provided scripts.

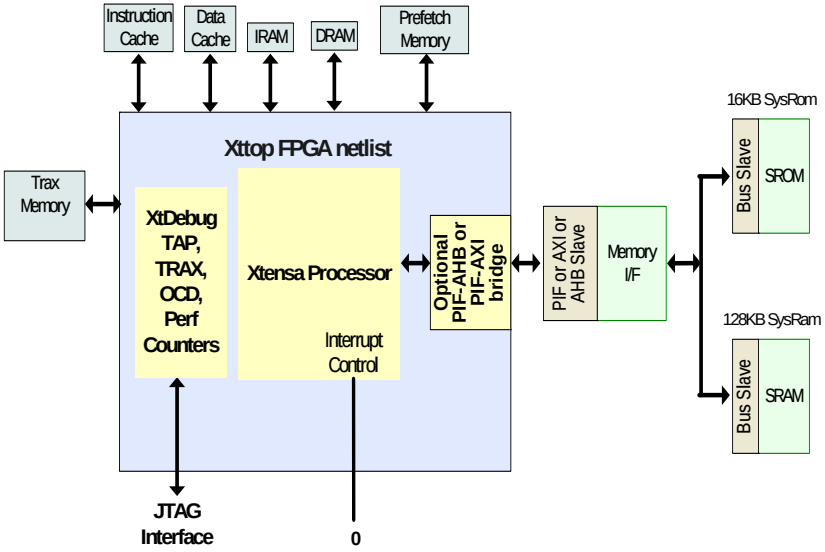


Figure 31–173. Sample Xtensa System

The FPGA Hardware Emulation Package main components are:

- Xttop FPGA netlist
  This is the Xttop netlist targeted to the Xilinx Virtex Family of FPGAs.

- FPGAMain.v
  Sample system with the Xttop module instantiated along with all local memories and a bus interface.
- Bus Interface
  Sample bus interface as part of a reference system.
- FPGA Memory Models
  Models that utilize the block rams supported by Xilinx FPGAs.
- FPGAMain.xst
  Sample xst script for the sample system.

32. Low-Power SOC Design Using Xtensa Processors

The Xtensa processor has been designed with numerous features to minimize power consumption, including:

- Extensive multi-level clock gating, which disables register clocks when individual registers are not active.
- The ability to disable both instruction and data memories when they are not needed, without loss of performance.
- Optional data gating on unused TIE semantics and both instruction and data memories to prevent unnecessary toggles
- An available power control module (PCM) to support power-shut-off (PSO) of separate power domains (processor logic, debug logic, and memories) to reduce leakage power consumption, which is described in Chapter 33, “Power Shut-Off (PSO)” on page 569.
- An available L0 Loop Buffer, which will capture and run small zero overhead loops without having to enable the larger instruction memories.

In addition, the designer can minimize power by making judicious configuration choices. For instance, the designer may choose smaller and fewer memories, the designer may avoid configuration options that require large amounts of logic that are not needed for an application, and the designer may configure extra instructions or design TIE instructions that more effectively execute an application so that the processor can spend more time in sleep mode. The best choices for power vary with the application.

32.1 Saving Energy Through Instruction Extensions

Lowering power and energy consumption is one of the central reasons for using configurable processor cores and instruction-set extension. By adding appropriately tailored instructions to the processor’s ISA, the processor executes the target application code in fewer cycles. As a result, the processor can operate at a lower clock frequency, which in turn reduces power dissipation and energy consumption.

Table 32–121 shows the energy reduction achieved through ISA extension for four common embedded tasks: taking the dot product of a matrix, AES encryption, Viterbi decoding, and an FFT. The table shows energy reductions ranging from 2x to 82x. In some cases, the mW/MHz number grows through ISA extension because gates must be added to extend the ISA. However, ISA extensions lower the processor’s required operating frequency, which more than compensates for the power increase due to the additional gates. Overall, the processor’s energy usage drops.

Table 32–121. Energy Reduction Through ISA Extensions

Application		Reference Processor Configuration (Normalized)	Optimized Processor Configuration (Relative to Reference)	Reduction in Energy Usage from ISA Extensions
Dot Product	Clock Cycles	1	0.49	
	Area	1	1.44	
	Energy Used	1	0.49	2x less energy
AES Encryption	Clock Cycles	1	0.01	
	Area	1	2.00	
	Energy Used	1	0.01	82x less energy
Viterbi Decoder	Clock Cycles	1	0.027	
	Area	1	1.20	
	Energy Used	1	0.03	33x less energy
FFT	Clock Cycles	1	0.04	
	Area	1	1.50	
	Energy Used	1	0.04	22x less energy

32.2 Saving Power Through Clock Gating

The Xtensa processor provides two levels of clock gating. The first level of clock gating is based on global conditions. For instance, the WAITI option allows the processor to enter a sleep mode that turns off the clocks to almost all of the registers in the design. WAITI requires interrupts to be configured, because an interrupt wakes the processor from sleep mode. If interrupts are not configured, the `RunStall` signal can still be used to save power by allowing external logic to stall the processor pipeline and turn off the clock to many of the processor's registers. Both WAITI and RunStall are described in more detail in this chapter.

Other global conditions that allow the first level of clock gating to save power include instruction-cache line fills, and various conditions in the load/store units.

The second level of clock gating is functional clock gating, which is much more local to specific registers in the design. In many cases specific register clocks can be gated based on local conditions in the processor. Even though the processor as a whole is active, portions of the design can be deactivated and reactivated as needed to save power using this method. Choosing Functional clock gating creates many smaller branches in the clock tree that are activated by individual enable signals. The Xtensa processor employs extensive clock gating for hardware added through designer-defined TIE constructs, which makes such extensions extremely power efficient.

For maximum power savings, both levels of clock gating should be enabled. However, if the designer only desires one level of clock gating due to design-flow restrictions, then just one (or neither) of the clock gating levels can be chosen. In general, if the processor will have long periods of inactivity, then the `WAITI` or `RunStall` options will save the most power. If the processor is mostly active, then the functional clock gating will save the most power. The designer must make choices based on the target application(s).

32.3 Optimizing Memory Power

The selection of local memories for both instruction and data can have an important effect on power consumption, and involves many tradeoffs.

In general, using instruction RAM and data RAM is more power efficient than using instruction caches and data caches of a similar size, as there are no tag arrays and no tag comparisons to determine a cache hit or miss. This can work especially well if the code instructions and data fits completely in the local memories. If the instructions or data does not fit completely in a reasonably sized instruction RAM and data RAM, and the contents of these memories have to be dynamically managed, then an instruction cache or data cache can be a good choice. In general, reducing the number of cache ways will also reduce the power and energy consumed for a given application, assuming that the cache hit rate does not suffer too much.

Memory choices must be guided by a combination of performance analysis and simulation, as well as area and power analysis.

The Xtensa processor includes a number of power optimizations, which are useful when performance requirements are such that there are multiple instruction local memories or multiple data local memories. This is especially true on the instruction side, where Xtensa processors can often figure out that it is executing out of a specific instruction RAM, instruction ROM, or cache way, and can disable all the instruction local memories that it does not need. Similarly, if there are multiple data memories, such as data RAMs and a data cache, Xtensa processors often know that it is not accessing certain memories, and save power by not enabling them.

Your core will include the local memory optimizations described above if you configured it with the "Region Protection Unit" (RPU) option, either with or without address translation (but not the full MMU option). Despite these optimizations, the data RAM memories may still need to be speculatively enabled. In general, to ensure maximum power savings, hardware designers should differentiate the addresses of PIF memories and data RAMs. Specifically, you should design the PIF memory map so that the physical addresses of PIF memories and devices are not placed in the same 512 MB region as the data RAM address ranges that are specified in the processor configuration. To further

reduce the number of speculative accesses to data caches, write your firmware to set the data access mode to Bypass for the regions that contain data RAMs whenever possible.

If Dynamic Cache Way Usage Control is configured, cache ways can be selectively disabled. During periods of operation where it is determined that less cache is needed for performance reasons, power savings can be realized by using this feature to gracefully shutdown any number of Instruction-Cache or Data-Cache ways by marking them as not usable. Power savings will be realized by a reduction in the amount of tags and cache ways (and hence memory arrays) accessed in parallel on any given access.

32.4 Configurable Data Gating

In both configurable instruction set options and user-defined TIE instructions, the output of register files and states fan out to several parallel TIE semantic logic blocks. At any given time, not all of these semantics are in use. If the semantic data gating option is configured, data gates with the appropriate enables are inserted in front of these semantic logic blocks to prevent unnecessary toggles. There are two configuration options:

1. If "all" is selected, then all user-defined TIE semantics and a number of Cadence -defined TIE semantics are data gated.
2. If "user_defined" is selected, then only TIE semantics identified by the user with the data_gate property are gated, as well as a Cadence-determined optimal list of core TIE semantics.

Although the Xtensa core will lower memory enable signals when a particular memory is not in use, the outbound address and data lines will still toggle. Data gating these signals can save idle-cycle memory dynamic power. If the memory data gating option is configured, data gates with the appropriate enables will be inserted on all instruction and data memories. Whenever the addresses to the different memories will be maintained low, and the write data will be either maintained low or at its previous value. This power reduction feature is most effective when there are several local instruction or data memories. The extra data gating may cause a slight decrease in operating frequency.

32.5 Power Shut Off (PSO)

As process geometries continue to shrink, care must be taken to reduce leakage power when portions of the Xtensa subsystem are idle. If PSO is configured, the Xtensa subsystem is divided into voltage domains that can be powered down either by the processor core or the external user. A power control module (PCM) logic block is provided to control the necessary shutdown and wakeup sequences, while configuration-specific UPF/CPF files are provided to drive the EDA tools. PSO is a feature-controlled portion of the product. See Chapter 33 for further details.

32.6 Loop Buffer

To further reduce instruction memory power consumption, a level-zero Loop Buffer of up to 256 bytes can be configured. If small pieces of code are executed repeatedly in a zero overhead loop (for example, loops set up with LOOP, LOOPGTZ, or LOOPNEZ instructions), these instructions can be captured and executed out of the loop buffer while the other instruction memories can be powered down. If the target application has lots of code that can be executed as part of the body of a zero overhead loop, which happens frequently in data-plane applications, then the loop buffer can have substantial savings on overall instruction memory power.

The loop buffer only captures data once a loop is encountered, so that the buffer is not continuously capturing data in non-loop back type code. Additionally, the loop buffer can itself be turned off for further power savings.

Bit 0 of the L1 Memory Controls register (MEMCTL[0]) is used to control the Loop Buffer. When set to 0, the Instruction Buffer is not used, and the logic is largely clock gated for power savings. When set to 1, the Loop Buffer is actively trying to capture loop body instructions, and trying to execute out of the Loop Buffer whenever possible.

Table 32–122. Loop Buffer Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
MEMCTL[0]	1	1	L1 Memory Controls	R/W	97

1) Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions.

Under some conditions the loop buffer may stop capturing instructions of a loop. Some conditions, like cache misses, might only occur in the first iterations of a loop. In this case, the loop buffer may be able to capture instructions on subsequent iterations of the loop. The events which prevent a loop buffer from capturing instructions include:

- an instruction cache miss
- an uncached instruction fetch
- any store to instruction RAM
- an ISYNC instruction
- any instruction cache invalidate instruction (IHI, III, IHU, IIU)
- any instruction TLB instruction (IITLB, PITLB, RITLB0, RITLB1, WITLB)
- writing to LBEG, LEND, RASID or PTEVADDR registers
- writing 0 to MEMCTL[0]

32.7 Wait Mode

The Xtensa interrupt option adds the PWaitMode port and WAITI instruction to the processor. When the WAITI instruction completes, it causes the processor to set the interrupt level in the PS.INTLEVEL register to the value encoded in the WAITI instruction. The processor then waits for all of the processor memory interfaces, TIE queues, and TIE lookups to become idle. After all of its interfaces become idle, the processor asserts PWaitMode and suspends operations until a non-masked interrupt occurs.

If the global clock gating option is enabled, then the majority of the clocks in the processor will be switched off while PWaitMode is asserted. If global clock gating is not enabled, then processor operations will still be suspended while PWaitMode is asserted, but there will be no power savings inside the processor.

Inbound-PIF requests are not considered processor operations and can continue while the processor is in wait mode. The clock network to the inbound-PIF logic is gated by a separate set of conditions to allow the inbound-PIF logic to continue functioning while the processor is in wait mode.

If PIF write responses are not configured, the processor core considers the PIF interface to be idle once all pending write requests have been accepted on the PIF interface. Configurations with the AHB-Lite or AXI bus bridge have additional delays for each write request. These delays are accounted for while generating the PWaitMode signal. It does not get asserted until all writes have been processed by AHB-Lite or AXI bus bridge. If PIF write responses are configured, the PIF interface itself does not become idle until write responses return from the AHB-Lite or AXI bus.

The processor exits PWaitMode once the processor receives an interrupt.

Note however there is no guarantee that the PWaitMode output will be de-asserted before the first interrupt vector instruction fetch request goes to memory. So for instance, the PWaitMode output should not be used to turn on other parts of the SOC that are responsible for responding to instruction fetch requests. The assertion of one of the BInterrupt inputs should be used for this purpose, as the instruction fetch will begin in a well defined manner following BInterrupt being asserted (see Appendix A.8).

32.8 Global Run/Stall Signal

The RunStall signal can be used to save power without using interrupts, as is done in wait mode. RunStall shuts off most—but not all—of the processor's clock network, assuming first-level clock gating is configured. In particular, the clock to the PIF logic will not be gated off, nor will the clocks to some internal logic needed to complete any cache-line fills that are in progress. In general, local-memory enables will be deactivated, except when cache-line fills are completing or when inbound-PIF requests to instruc-

tion or data RAMs are active. Because the RunStall signal switches off fewer clocks than in the WAITI interrupt-based power-saving scheme, power savings in general will be less. However, this technique does not require the use of interrupts.

RunStall can be activated at any time, but activating this signal stalls the internal processor pipeline. Thus, the designer is responsible for determining when it is appropriate to activate the external RunStall signal.

32.9 TIE Queue and TIE Lookup Stall

When an Xtensa processor stalls due to one of the following conditions:

- A TIE output queue being full
- A TIE input queue being empty
- A TIE lookup not giving the Rdy signal

the processor will enter a low power mode that is roughly equivalent to the low power mode caused by the Global Run/Stall signal. In some cases an extra stall cycle will occur to allow the instruction memories to be disabled and then re-enabled when the stall condition goes away. This extra stall cycle occurs when the instruction memory is being enabled during the stall, and the stall condition lasts for sixteen cycles or more.

32.10 Traceport Enable

The PDebugEnable signal can be used to save power in configurations with the Traceport and functional clock gating. When the Traceport output from the Xtensa core is to be consumed, as in an active debug session, PDebugEnable must be set high. At other times, PDebugEnable can be set low. If the configuration has functional clock gating, setting PDebugEnable low gates off clocks to the flip-flops in the Traceport logic. This eliminates toggles in the Traceport logic and in the Xtensa Traceport interface, preventing wasted power.

PDebugEnable is registered once inside the Xtensa core and the registered version of the signal is used to gate the clock to the flip-flops in the Traceport logic. Due to the pipelining of trace data within the processor, the Traceport output will be undefined for several cycles after PDebugEnable is asserted. In the cycles where data is undefined, PDebugStatus[1:0] will be "00," implying a bubble and PDebugData[31:0] will indicate "All other pipeline bubbles."

PDebugEnable can be changed dynamically while the processor is running. It is the designer's responsibility to control this interface using logic external to the processor. The signal can be tied high to always enable trace output, tied low to prevent any trace output, or controlled by external logic that can dynamically change the value.

The preceding refers to a user of the Traceport that is external to Xtensa. Note that there are internal users of the Traceport also such as the TRAX and the Performance Monitor. The latter also assert the `PDebugEnable` signal when the functions (i.e. PC tracing or performace monitoring) are enabled. This is achieved via an OR gate inside Xtensa that combines the respective `PDebugEnable` signals.

33. Power Shut-Off (PSO)

As process geometries shrink, leakage power consumes a larger portion of the total power budget. To substantially reduce leakage power, options are provided during processor configuration which will:

- Instantiate a power control module (PCM) in the Xtmem level of design hierarchy
- Specify the number of power domains within the design and their operation via industry standard power format files

33.1 PSO Flow

Each power domain consists of blocks of logic that operate at the same voltage and that can be concurrently shut down or powered back on. By taking inputs from both the outside world and software running on the core, the PCM module provides the necessary signals in the correct sequence to properly control each power domain. This chapter describes the software and hardware flows available to control this process.

33.1.1 Configuration Options and Restrictions

The following PSO configuration options are supported:

- A single large power domain with no state retention (referred to as "Single").
- Three separate power domains (referred to as "Core/Debug/Memory") with either full or no Core domain state retention.

If either PSO configuration is selected, the following configuration options are required:

- At least one high level interrupt (level 2 or higher) of type ExtLevel.
- On-chip-debug (OCD) must be enabled.
- If a system uses a bridge that may temporarily hold inbound PIF requests/responses, the Write Response option is required to ensure no PIF requests/responses get stuck in the bridge during PSO. External requestors may make use of the Write Response option to provide end-to-end check that a write has completed
- If selecting Core/Debug/Memory power domains and full Core state retention, the register files must be implemented using flip-flops (latches are unsupported).

Note:

- If the processor is configured with TIE queues interfaces, ensure that no data remains in processor internal buffers for the TIE queues before starting the power-down process. See Section 33.1.2 for more information.
- If the processor is configured with an inbound PIF interface, the external system must not post any new inbound transactions to the core after the core has indicated it is ready to be powered down. See Section 33.1.2 for more information.
- The "Single" power domain configuration is an entry-level technique that does not allow for graceful processor shutdown. It is intended for use when the entire Xtensa processor subsystem has a single master in control of subsystem use. All discussions in this chapter assume the Core_Debug_Memory option unless otherwise noted.

Power Domain Descriptions

The Xtmem logical hierarchy and the available power domains are illustrated in Figure 33–174. Note that if the "Single" domain is selected, the core, memory, and debug power domains are combined into a single power domain (as illustrated by the dashed purple line).

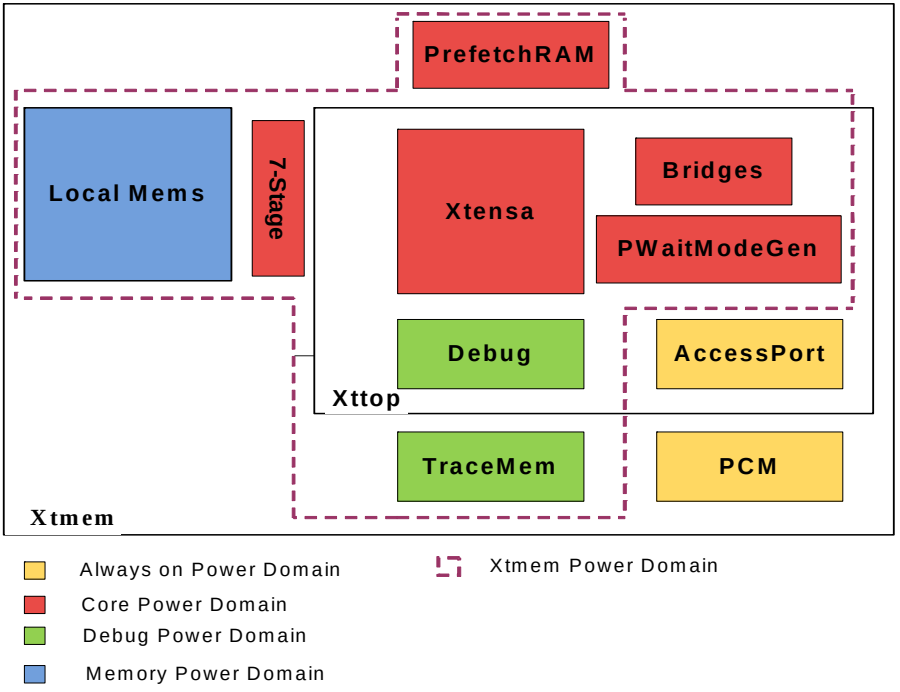


Figure 33–174. PSO Support

The external system communicates with the Power Control Module (PCM) through the Access Port Module registers PWRCTL (Power and Reset Control Register) and PWRSTAT (Power and Reset Status Register), which are in the always-on power domain. The Access Port and PCM function together in the PSO flow. Externally, PSO is controlled by either the JTAG, APB port, or individual signals from an external system. The debug module PSO can also be controlled by the core using WER and RER instructions.

JTAG, APB, and the direct External PSO interface (consisting of the PsoExternal{Proc,Mem,Debug}Wakeup signals) interface to the Access Port, give the command that a shut-down or wake-up is requested. For the three-domain configurations, shut-down of the Core domain requires participation of code running on the Xtensa processor, in addition to de-asserting all wake-up requests. The method of communicating any external shut-down request to the processor software is not specified; as an example, an additional external interrupt pin can be configured for this purpose. If state retention is configured, an additional level 2 or higher interrupt also needs to be configured, left enabled but de-asserted on shut-down, and asserted on wake-up. Upon power-up, if state retention is not configured, the Xtensa core goes to the Reset handler; if state retention is configured, the Xtensa core wakes up in WAITI mode (executing the WAITI instruction), and this (level 2 or higher) interrupt is used to get the processor running again.

The Access Port communicates with the PCM, which starts its state machine and supervises the shut-down/wake-up process by controlling the clocks and power gating signals. Hence, both these modules must always be powered on so the external system can dynamically control the PSO of the Xtensa subsystem.

The core domain is powered ON during normal operation. Following are the use models to power down the core domain.

- Memory Save-and-Restore: Just before removing power, software saves the core domain state – specifically, the processor architectural state – to memory. Upon powering the core back up, software restores the state from memory.
- Full Retention: When the core domain is powered down, the state of all its flops is retained in the retention registers. When it is powered back on, the state of all its flops is restored. Compared to memory save-and-restore, this approach enables fast wake up at the cost of increased area.
- Stateless: The core power domain is powered down without retaining the state. This mode is useful when the processor is powered down after completing a task that need not maintain state for its next invocation – such as audio decoding of a stream (though usually not decoding of a single frame in a compressed stream). When the processor wakes up in response to arrival of a new task, the previous state is irrelevant; it can be treated as if it is powered on for the first time. This stateless option is the only one supported if the processor is configured for single-domain PSO.

Whether the core domain can be powered down without retaining the state depends on the use of the Xtensa core in the system context. When the state must be saved, the decision to use memory save-and-restore, or full retention depends on the following factors: the leakage power savings, additional area / power cost, and verification complexity.

Description of PCM Control from the Xtmem Level

Each of the three preconfigured power domains (Core, Debug, and Memory) has three separate user-controllable methods to request a wakeup: via the JTAG interface, via the APB interface, and via the "External" interface, which is a single-bit input wire that can be driven by hardware outside the Xtmem module interface. For safety, a power domain will only begin the shutdown sequence if *all three* wakeup requests are set low. This is to ensure that if some external agent is still using the power domain, it is not shutdown simply because a single agent no longer requires it.

The Debug and Memory power domains have a simple control scheme. Upon asserting PcmReset, these power domains are on. When all three {JTAG, APB, and External} wakeup signals are low, then these domains begin the shutdown sequence. Afterwards, if any one of the JTAG, APB, or External signals goes high, then the specified domain powers back on.

The Core domain is slightly different: software is another method to initiate the shutdown process – usually by calling `_xtos_core_shutoff()`. (This can happen, for example, in response to an interrupt.) Once the core is shut down, if any one of the JTAG, APB, or External signals goes high, then the core domain powers back on. Even if all JTAG, APB, and External signals are low, the core will not shut down unless it executes the appropriate shutdown sequence in software. Conversely, if the core executes `_xtos_core_shutoff()` and one of the JTAG, APB, or External signals to stay powered up is still high, the core will not shut down.

In the single-domain case, there is no requirement for software involvement in the shutoff process. An Xtensa core shutoff happens unconditionally based on deassertion of the external wakeup hardware signal. However, Cadence recommends that there be some system-level co-ordination to ensure graceful shutoff. Indeed, the reference one-domain PSO flow (supplied with the RefTestbench) causes the core to go into WAITI mode before shutoff.

33.1.2 Core PSO Flow Description

The PSO mechanism for the Xtensa core consists of the following three parts: the power down process, the processor interface when powered off, and the power up process.

Power Down Process

The power down process has three steps: power down trigger, execution of power down routine, and hardware signal sequencing to turn off the power to the core.

1. Power Down Trigger: The decision to shut off the core may occur in the core itself (in software). Alternatively, it may occur in an external agent, which must communicate its decision to the core (such as, using an interrupt or other mechanism) so that the core can invoke the power down routine, such as `_xtos_core_shutoff()` or `_xtos_core_shutoff_nw()`.
2. Power Down Routine: For details on the software power down sequence, refer to the description of the `_xtos_core_shutoff()` function in the *Power Shut-Off* section of the *Basic Runtime (XTOS)* chapter in the *Xtensa System Software Reference Manual*.

   Note that it is up to the caller of this function to disable all interrupts except for the one used to resume execution after core wake up.

   The waiti instruction will assert a PWaitMode signal after completion of activity on all Xtensa interfaces. For configurations with slave PIF interface configured on the Xtensa core, the PWaitMode signal is asserted only after the current inbound PIF transaction is complete. The PsoShutProcOffOnPWait output signal indicates to PCM that the Xtensa processor has begun software shutdown routine. The assertion of both PSOShutProcOffOnPWait and PWaitMode signals indicates that the processor is ready to be powered down. Any inbound PIF transactions posted to the processor after both these signals have been asserted, and before the core powers down, may be lost, or may put the processor in an invalid state.Once PSOShutProcOffOnPWait is asserted, it is the system's responsibility to keep the External Wake-up signals asserted until any pending Inbound PIF or interrupts have been completed.
3. Powering Down Hardware: The power down request for the core power domain is generated when both PsoShutProcOffOnPWait and PWaitMode signals are asserted. This initiates the shutdown sequence within the PCM block; during this time, any new interrupt is disregarded and POReqRdy is isolated low for configurations with inbound PIF.

In a core with full state retention, when the processor is powered up at wake up, its architectural state will be restored just as it was before power down. That is, the processor will be in the WAITI state. To bring the processor out of WAITI mode, a power up interrupt must be provided. Thus if WAITI *n* was used, which disables interrupts up to priority *n*, an interrupt of priority level *n*+1 or higher is required to wake-up from the WAITI state.

Interface Behavior when Powered Off

The interface behavior during power down is based on the isolation state of interfaces.

- PWaitMode is retained as asserted.
- Inbound PIF: After power down, POReqRdy is retained as deasserted. The processor will not accept inbound PIF requests while powered down. The external system should buffer those transactions.
- Interrupts: Interrupts are ignored when the processor is powered off.
- Memories: Memory enables are retained as deasserted so as to save memory power if the memory domain remains powered on.
- TIE input and output queue control interface signals are retained as deasserted.
- All Xtensa/Xttop output signals retention values can be found in the UPF/CPF files.

33.1.3 Processor Interfaces and System Behavior

System designers must plan for the behavior of their system when powering down an Xtensa processor. The power-down process described in Section 33.1.2 is designed to put the processor in a known state, with no outstanding transactions on the PIF or memory interfaces. In addition, the system designer may need to use application level protocols to ensure that no data is lost on interfaces into the Xtensa core when it is powered down. This is especially true for optional interfaces that directly connect to the external system, and have internal state that is not architecturally visible. These interfaces are listed below.

Inbound PIF

As noted above, when the processor completes its power-down routine, it should execute a `WAITI` instruction to ensure that all memory transactions have completed. It will then signal the system that the processor is ready to be powered down by asserting both `PSOShutProcOffOnPWait` and `PWaitMode` signals. However, `PWaitMode` does not usually prevent requests from the system to the Inbound PIF interface.

For correct operation, the external system should not post any inbound PIF requests to the processor after the processor has asserted `PSOShutProcOffOnPWait`, indicating that it is preparing to power down. After the processor asserts both `PSOShutProcOffOnPWait` and `PWaitMode` signals, any additional inbound PIF requests posted before power down may be lost, or may leave the processor in an invalid state. Once the processor has powered down, it will deassert `POReqRdy` and will not accept inbound PIF requests. The external system should then buffer those inbound PIF transactions.

TIE Output Queue

The protocol of a TIE output queue interface allows an instruction that pushes data to the queue to commit, even if the external FIFO is full. In that case, the data is stored in an internal buffer until it can be pushed to the external FIFO. That internal buffer is not architecturally visible, so it will not be saved or restored by PSO software routines.

The EXTW instruction ensures that any data pushed to TIE output queues commits, and is accepted by the external FIFO. Alternatively, when the processor completes its power-down routine, it should execute a WAITI instruction. This instruction waits for all of the processor memory interfaces, TIE queues, and TIE lookups to become idle. After all of its interfaces become idle, the processor asserts PWaitMode. This ensures that any committed push operations to a TIE output queue are completed to the external FIFO.

TIE Input Queue

A TIE input queue is a *speculative* interface, in which an instruction may pop data from an external FIFO, and then that instruction might not complete. For instance, the instruction could be aborted by an interrupt or exception. In such cases, the processor buffers the TIE input queue data in an internal buffer so that no data is lost. This internal buffer is not architecturally visible, so it will not be saved or restored by PSO software routines. The TIE input queue internal buffer is not affected by an EXTW or WAITI instruction.

The internal queue buffer contents will only be saved during power-down and restored in a core with *full retention*. Otherwise, in a stateless core, or one using memory save and restore, any speculatively popped TIE input queue data will be lost on power down. The internal buffer data is not retained if the core is reset.

In order to power down and power up a processor configured with a TIE input queue, without losing any data on the interface, system designers should either use PSO with full state retention, or use higher level protocols to ensure that no data remains in the TIE input queue buffers.

33.1.4 Power Up Process

The power up process consists of three steps: power up trigger, powering up hardware, and execution of the power up routine.

1. Power Up Trigger: The power up process for the Core power domain is initiated by driving the PsoExternalProcWakeup signal to logic high, or by setting one of the CoreWakeup bits in the Access Port PWRCTL register using the JTAG or APB port.

   The power up process for the Debug power domain is similar, except that the processor itself may also access the PWRCTL register using WER and RER instructions.

For details on the PWRCTL register, see the *Power, Reset and Clocks* chapter of the *Xtensa Debug Guide*.

2. Powering up Hardware: When the processor is powered on, depending on the selected 'PSO Core Retention' configuration option, it will be in the following state:
   - No State Retention: It will be in a Reset state. From there it jumps to the reset handler.
   - Full State Retention: It will be in the WAITI state-it went into during the `_xtos_core_shutoff()` routine (or equivalent user function). The core now needs to be triggered by an interrupt from the system to complete the rest of the `_xtos_core_shutoff()` function. As most of the interrupts are disabled during power shutdown, be sure to use an interrupt that remains enabled before powering down.
3. Power up Routines: For details on the software wake-up sequences, refer to the *Normal Power Shut-Off and Wake-Up Sequence* subsections in the *Power Shut-Off* section of the *Basic Runtime (XTOS)* chapter, in the *Xtensa System Software Reference Manual*.

33.2 PSO Signals and State Transitions

The signals available at the Xtmem module boundary to interface to the PCM depend on how many power domains were configured at build time. The two available options are "single" and "core/debug/mem".

33.2.1 Single Power Domain

If you select the single domain structure, then the entire Xttop, Debug, and Memory subsystem is treated as one monolithic power domain. Only the PCM and Access Port modules stay always-on. State retention is not supported, nor can the core gracefully shut off. For a single domain structure configured this way, the Xtmem boundary signals for PSO are shown in Table 33–123.

Table 33–123. Xtmem Boundary Signals

Name	Width	Direction	Description
PcmReset	1	Input	Reset signal for the PCM module; this signal must be asserted first at system wakeup to guarantee power is available to all power domains; this as well as all other input reset signals must be driven for at least 10 cycles of the slowest clock period. After initial wakeup this signal does not need to be asserted again.
PsoExternalMemWakeup	1	Input	Main control point for single power domain; when driven logic high, the Xtmem power domain will begin its wakeup sequence or stay in the on state; when driven low, the Xtmem power domain will shut down. Once either the shutdown or wakeup sequence is initiated, it will run to completion.
PsoDomainOffXtmem	1	Output	Identifies state of Xtmem power domain; immediately after PsoExternalXtmemWakeup is driven low, this signal will be driven high to indicate that the domain is off and should not be interacted with. If PsoExternalXtmemWakeup is driven high, only after the full wakeup sequence has completed will PsoDomainOffXtmem be driven low

Given that there is only one power domain, the allowed states are straightforward – these transitions are controlled with the PsoExternalXtmemWakeup signal:

- For normal operating mode, the Xtmem power domain is on
- When in shutdown operating mode, the Xtmem power domain is off

Cycling the PcmReset signal during system wakeup will start the Xtmem power domain in the on state (denoted by X_1 in Figure 33–175), the allowed transition table is shown in the following figure.

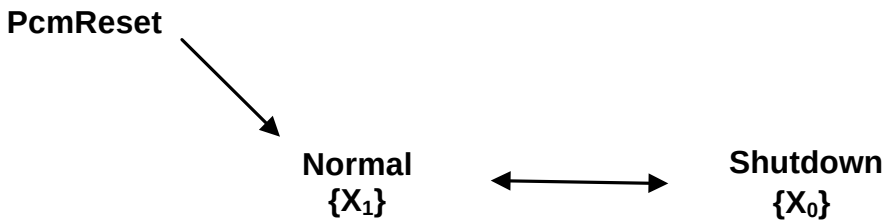


Figure 33–175. PcmReset Signal Cycle

33.2.2 Core/Debug/Memory Power Domains

If you select the core_debug_mem domain structure, the Xtmem subsystem is partitioned into three separate power domains consisting of local memories, debug logic, and core processor logic. Only the PCM and Access Port modules stay always-on. Each domain can be controlled independently, with support for a graceful shutoff of the processor core. Additionally, full state retention for every flop in the “Core” power domain can be configured. If debug logic is not configured, the “Debug” power domain will not be present. The Xtmem boundary signals configured for core/debug/memory for PSO is shown in Table 33–124.

Table 33–124. Xtmem Boundary Signals for Core/Debug/Memory Power Domains

Name	Width	Direction	Description
PcmReset	1	Input	Reset signal for the PCM module; this signal must be asserted first at system wakeup to guarantee power is available to all power domains; this as well as all other input reset signals must be driven for at least 10 cycles of the slowest clock period. After initial wakeup this signal does not need to be asserted again.
PsoExternalProcWakeup	1	Input	External control point for the Core power domain; when driven logic high, the Core power domain will begin its wakeup sequence or stay in the on state; when driven low, the external world indicates it no longer needs the Core power domain. Only when all Pso*ProcWakeup signals are low, and the core receives a shutdown request interrupt, will the Core power domain shut down. Once either the shutdown or wakeup sequence is initiated, it will run to completion.
PsoExternalMemWakeup	1	Input	External control point for Memory power domain; when driven logic high, the Memory power domain will begin its wakeup sequence or stay in the on state; when driven low, the external world indicates it no longer needs the Memory power domain. Only when all Pso*MemWakeup signals are low will the Memory power domain shut down. Once either the shutdown or wakeup sequence is initiated, it will run to completion.
PsoExternalDebugWakeup	1	Input	External control point for the Debug power domain; when driven logic high, the Debug power domain will begin its wakeup sequence or stay in the on state; when driven low, the external world indicates it no longer needs the Debug power domain. Only when all Pso*DebugWakeup signals are low will the Debug power domain shut down. Once either the shutdown or wakeup sequence is initiated, it will run to completion.

Table 33–124. Xtmem Boundary Signals for Core/Debug/Memory Power Domains

Name	Width	Direct-ion	Description
PsoDomainOffProc	1	Output	Identifies the state of the Core power domain; immediately after the core shutdown procedure begins, this signal will be driven high to indicate that the processor is off and should not be interacted with. Only after the full Core wakeup sequence has completed will PsoDomainOffProc be driven low.
PsoDomainOffMem	1	Output	Identifies the state of the Memory power domain; immediately after the memory shutdown procedure begins, this signal will be driven high to indicate that the Memory domain is off. Only after the Memory domain completes its wakeup sequence will PsoDomainOffMem be driven low.
PsoDomainOffDebug	1	Output	Identifies the state of the Debug power domain; immediately after the debug shutdown procedure begins, this signal will be driven high to indicate that the Debug domain is off and should not be interacted with. Only after the Debug domain completes its wakeup sequence will PsoDomainOffDebug be driven low.
PsoShutProcOffOnPWait	1	Output	Serves as an early warning indicator that the processor is about to begin its shutdown procedure and should no longer be interacted with. Once this signal and PWaitMode are both driven high, the PCM module will begin the shutdown sequence for the Core domain. Once the processor has completed its wakeup sequence, this signal is driven logic low again

The three power domains of Core, Memory, and Debug can be either on or off, which makes for eight possible functional states. The possible combinations and corresponding state names are shown in Table 33–125.

Table 33–125. Functional States for Core/Debug/Memory Power Domains

Operating Modes	Power Domain		
	Proc	Debug	Memory
Debug	On	On	On
Normal	On	Off	On
Bypass	Off	On	Off
Bypass + Mem	Off	On	On
Sleep	Off	Off	On

Table 33–125. Functional States for Core/Debug/Memory Power Domains

Operating Modes	Power Domain		
	Proc	Debug	Memory
Shutdown	Off	Off	Off
Invalid	On	On	Off
Invalid	On	Off	Off

The last two combinations are invalid since the Memory domain must always be on if the Core domain is on. The invalid states are not prevented in hardware, but they are specified as illegal in the UPF/CPF files, which would then cause error messages if they are entered during simulation.

A further constraint is that only one power domain is allowed to power up/down at a time. For example, to progress from the Debug state to the Shutdown state, the sequence would be: debug → normal → sleep → shutdown. There are no hardware restrictions in place to enforce these rules – they are part of the external interface definition. Cycling the PcmReset signal during system wakeup will start all power domains in the on state, corresponding to the Debug state in Table 33–125. The allowed state transitions are shown in Figure 33–176, with the Core, Debug, and Memory power domain states being represented as $\{C_X D_X M_X\}$, with X = 1 indicating the domain is on:

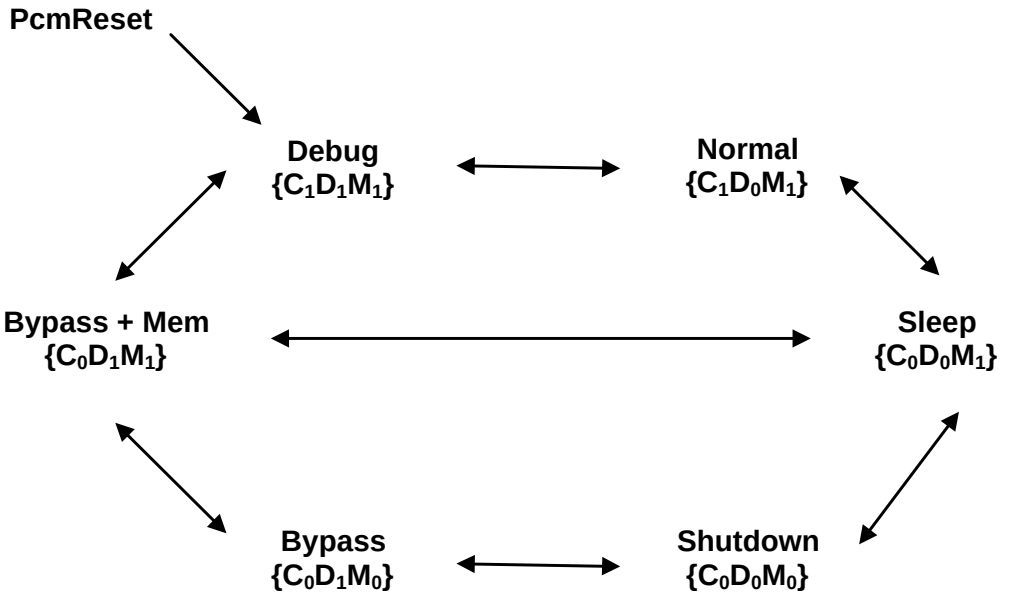


Figure 33–176. PcmReset Signal Cycle for Core, Memory, and Debug Power Domains

Note that due to the restriction that the Memory domain must always be on if the Core domain is on, there is a restriction on control sequencing as illustrated above: during shutdown, the core must be powered down before the memory. Conversely, during wakeup, the memory must be powered up before the core. The isolation requirements as specified in the CPF/UPF files were designed with these restrictions in mind.

33.3 PCM Control Signals

Within the Xtmem hierarchy, the PCM module has several signals that provide the necessary hardware interface to the power domain(s). These include: control signals for isolation cells on the output ports of a domain, enable and feedback signals for power switch control, and (optionally) control signals for the retention flops in the domain. The names of the signals with asterisks (*) indicate that there is one for each power domain, for example, PsoIsolate* means that the Debug domain will get `PsoIsolateDebug`, and the Core power domain will get `PsoIsolateProc`. The functionality of these signals is described in Table 33–126.

Table 33–126. PCM Signals

Port Name	Direction on PCM Module	Description
`Pso*Wakeup`	Input	Power domain control signals from the Access Port module. Each single bit signal represents a controlling agent and a power domain (i.e., PsoJtagDebugWakeup allows the JTAG interface to wakeup or shutdown the Debug power domain). A logic high value indicates that the domain should wakeup or stay on. A logic low value indicates that the user is fine with the domain shutting down.
`PsoDomainOff*`	Output	Each of these signals indicates when the respective power domain has been shut down. They are conservatively set – at the very beginning of a shutdown request, the respective PsoDomainOff* signal is driven high. The signal is only driven low again after the respective domain has completed the entire wakeup process.
`PsoIsolate*`	Output	Isolation gates need to be placed on power domain output ports if those ports can be off when a domain that reads those ports can be on. These gates can tie the value high, low, or maintain the last driven value (latch). The PsoIsolate* signals control these isolation cells. Due to potentially large fan out, these signals are given two clock cycles to change.

Table 33–126. PCM Signals

Port Name	Direction on PCM Module	Description
PsoBretainProc PsoSaveProc PsoBrestoreProc	Output	If configured at build time, state retention registers can be used to retain state in the Core power domain during shutdown. These signals are used to both save the value in the balloon latch on shutdown and to restore it on wakeup. Depending on the library, only one or two of these are needed – the others will be optimized away. Due to potentially large fan out, these signals are given two clock cycles to change. Single control-pin retention flops use the PsoBretainProc signal to perform both save (logic low) and restore (logic high) states, while dual control-pin retention flops use a high pulse of the PsoSaveProc signal to save, and a low pulse of the PsoBrestoreProc signal to restore state.
PsoRemovePower*	Output	Power switch cells are inserted during layout to connect the always-on global power (VDDG) to the switchable power domain power (VDD) – the PsoRemovePower* signals control these cells, one per power domain. The configuration of the power switch network (i.e. daisy chain, parallel columns, mother-daughter cells) is left to the user.
PsoRailFeedback*	Input	The PsoRailFeedback* signals are the return values from the propagated PsoRemovePower* signals in the power switch network. After asserting a PsoRemovePower* signal, the PCM state machine waits to see an asserted PsoRailFeedback* signal to know that it is safe to proceed to the next state in the shutdown/wakeup sequence.
PsoMemorySleep	Output	This signal supports more advanced memory hard macros that have light-sleep / deep-sleep functionality (LS/DS). Local memories are attached directly and exclusively to the Xtensa core. If the Core power domain is off but the Memory domain remains on, this signal can be used to put the local memories into the LS/DS state.
PsoCachesLostPower	Output	During the wakeup sequence, this signal notifies the processor core of the condition of the memory power domain - it is asserted only if the memory power domain was power cycled during the time the core was powered down; if the memory power domain remained on for the entire time the core power domain was powered down, than this signal remains deasserted.
PsoWakeupReset	Output	During the wakeup sequence, this signal notifies the processor what type of reset should be performed. If asserted, this means the core is coming out of a previous power down state. If deasserted, a full system reset should be performed.

33.4 Control Sequences

The following sections describe how an external agent to the Xtmem processor subsystem will initiate the power-down and power-up sequences for relevant power domains, as well as how the internal signals from the PCM are driven to affect these transitions.

33.4.1 Single Power Domain

Assuming that the Xtmem subsystem has been in the steady state on condition, a shutdown is initiated by the following:

1. The external user lowers the `PsoExternalXtmemWakeup` signal when the Xtmem subsystem is no longer needed.
2. All clocks to the Xtmem subsystem are stopped and the `PsoDomainOffXtmem` output is set to logic high to notify the external world that the subsystem is off.
3. `PsoIsolateXtmem` is driven high and given two cycles to fan out through the subsystem to all of the isolation gates on signal outputs.
4. `PsoRemovePowerXtmem` is driven high to all of the power switches in the subsystem.
5. The PCM module waits for the feedback signal `PsoRailFeedbackXtmem` to ensure that the power switches have all received the `PsoRemovePowerXtmem` signal.

The no-retention configuration single-domain shutoff is shown in Figure 33–177.

Note: As mentioned before, an Xtensa core shutoff happens unconditionally based on deassertion of the external wakeup hardware signal in the one-domain case. However, Cadence recommends that there be some system-level co-ordination to ensure a graceful shutoff. Figure 33–177 shows that the core is taken into WAITI mode before shutoff as an example of such co-ordination.

Note: The numbered row at the top of timing diagrams Figure 29-174 to Figure 29-183 denotes different phases or periods in the shutoff or wakeup sequence. Each phase is comprised of several clock cycles – whether CLK, JTCK or PBCLK – and represents a state that is meaningfully different from the previous state. The waveforms are not to (timing) scale and are used to demonstrate only the functional relationship between the various signals. To repeat, the time between two adjacent vertical dotted lines is not a single cycle, but a single period of multiple clock cycles.

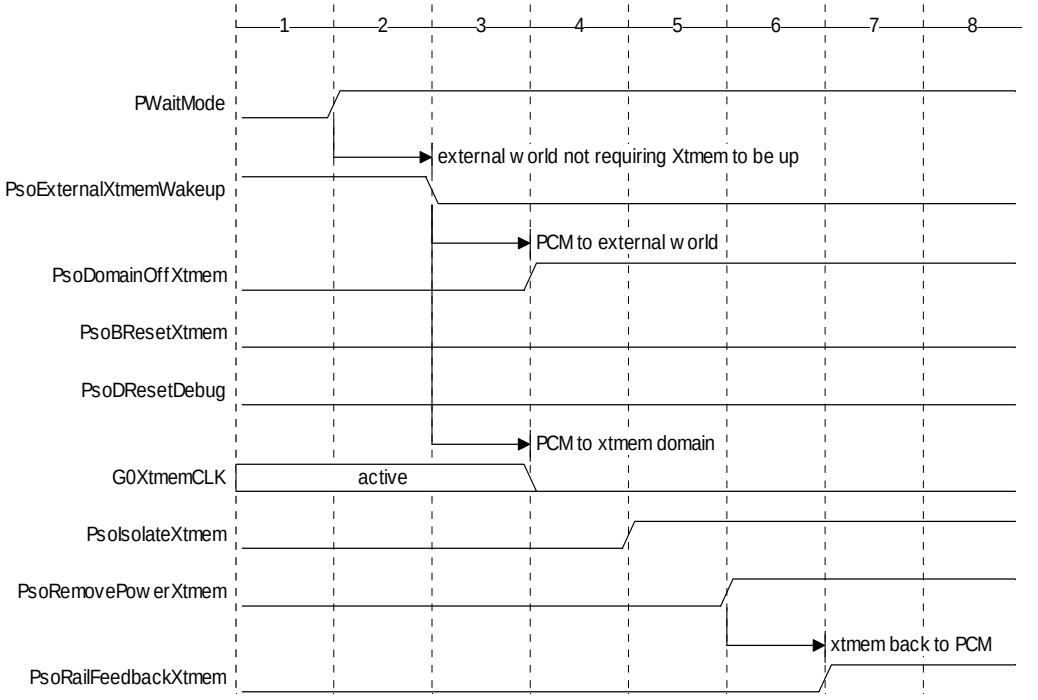


Figure 33–177. No-retention Configuration Single-Domain Shutoff

Assuming that the Xtmem subsystem has been in the steady state off condition, a wake-up is initiated by the following:

1. The external user raises the `PsoExternalXtmemWakeup` signal to begin Xtmem subsystem wakeup.
2. `PsoRemovePowerXtmem` is driven low to all of the power switches in the subsystem.
3. The PCM module waits for the feedback signal `PsoRailFeedbackXtmem` to also go low to ensure that the power switches have all received the `PsoRemovePowerXtmem` signal.
4. All system clocks (CLK, BCLK) are started and system resets (BReset, DReset) are driven high for at least six of the slower clock cycles. The resets are then lowered and the clocks are stopped again.
5. The `PsoIsolateXtmem` signal is driven low and given two cycles to fan out through the subsystem to all of the isolation gates on signal outputs.
6. All system clocks (CLK, BCLK) are permanently restarted.
7. `PsoDomainOffXtmem` is lowered to notify the external world that the subsystem is on.

The no-retention configuration single-domain wakeup as shown in Figure 33–178 is driven to affect these transitions.

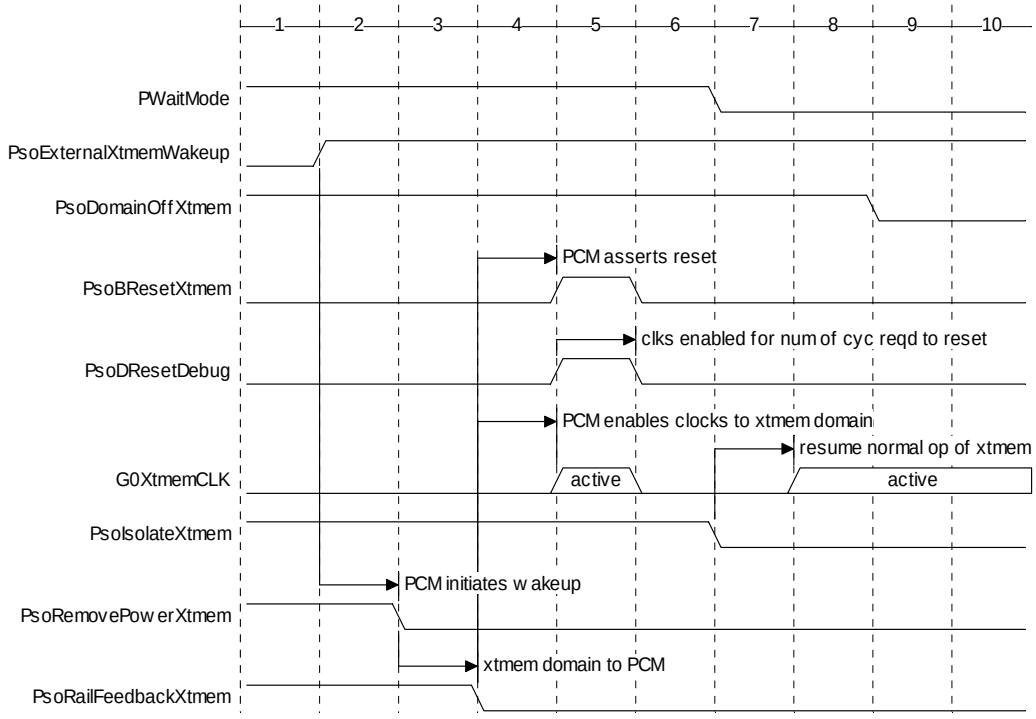


Figure 33–178. No-retention Configuration Single-Domain Wakeup

33.4.2 Core/Debug/Mem Power Domains

The shutdown and wakeup sequences for the Memory power domain are similar to those described in Section 33.1 for the Xtmem power domain and will not be repeated here. For clarity, waveforms are provided with the appropriate signal names.

The shutdown and wakeup sequences for the Debug power domain are also similar to those described for the Xtmem power domain but with one exception. The PCM does handshaking with the Debug Module before proceeding with shutoff. If the Debug Module is in use (for example, if TRAX tracing is enabled), shutoff will not be permitted. This is to prevent the Debug Module from suddenly turning off when for example you are debugging software through OCD. Thus, all debug functions must explicitly be turned off before shutoff can happen. In the interim, the FSM in the PCM controlling Debug domain power will be stalled. Following is a description for the complex case of Core domain shutoff.

Assuming the Core power domain of the Xtmem subsystem has been in the steady state on condition, a shutdown is initiated by the following:

1. All three of the Pso*ProcWakeup signals are driven low by all controlling agents (JTAG, APB, and external), indicating that none of the three agents still need the Xtensa core to be powered on. Additionally, a power down interrupt request is sent to the Xtensa core, initiating the software power-down routine.
2. When the Xtensa core completes the software power-down routine, it will set both `PsoShutProcOffOnPWait` and `PWaitMode` to logic high. Only when these two signals are high and the Pso*ProcWakeup signals are all low will the power down sequence begin.
3. All clocks to the Core power domain are stopped and the `PsoDomainOffProc` output is set logic high to notify the external world that the Core domain is off. Also, `PsoMemorySleep` is set logic high to implement LS/DS memory functionality.
4. `PsoIsolateProc` is set logic high and given two cycles to fan out through the Core power domain to all of the isolation gates on signal outputs.
5. (If full state retention is configured) `PsoBretainProc`/`PsoSaveProc`/`PsoBrestoreProc` signals are set to the correct values to save the state of the state-retention registers. These signals are given two cycles to fan out through the Core power domain.
6. `PsoRemovePowerProc` is set logic high to all of the power switches in the Core power domain.
7. The PCM module waits for the feedback signal `PsoRailFeedbackProc` to ensure that the power switches have all received the `PsoRemovePowerProc` signal.

The no-retention configuration processor domain shutoff is shown in Figure 33–179.

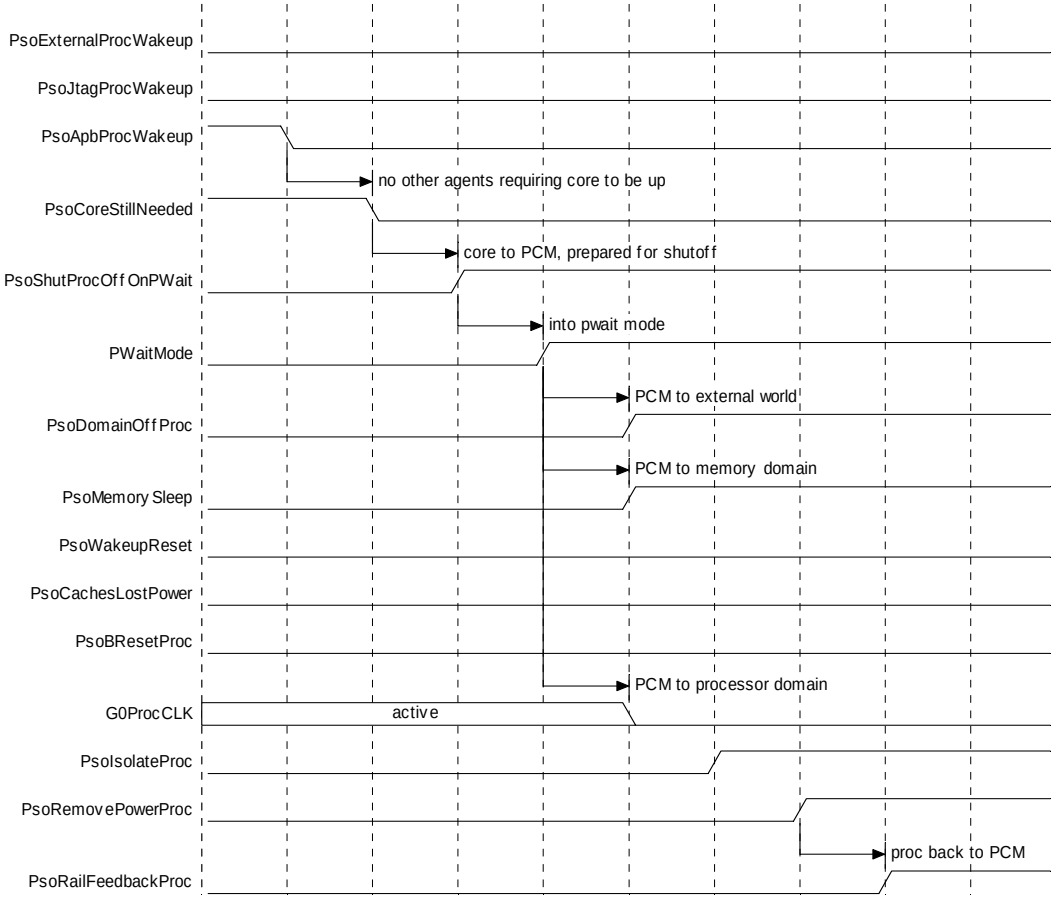


Figure 33–179. No-retention Configuration Processor Domain Shutoff

The no-retention configuration memory domain shutoff is shown in Figure 33–180.

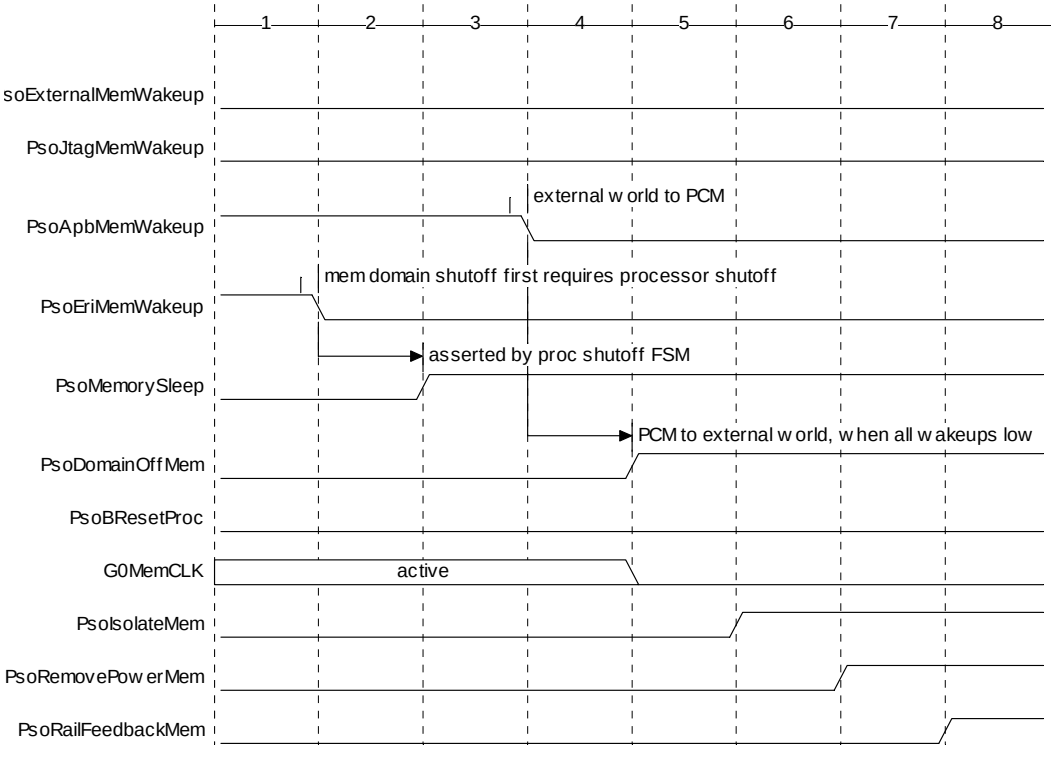


Figure 33–180. No-retention Configuration Memory Domain Shutoff

The no-retention configuration debug domain shutoff is shown in Figure 33–181.

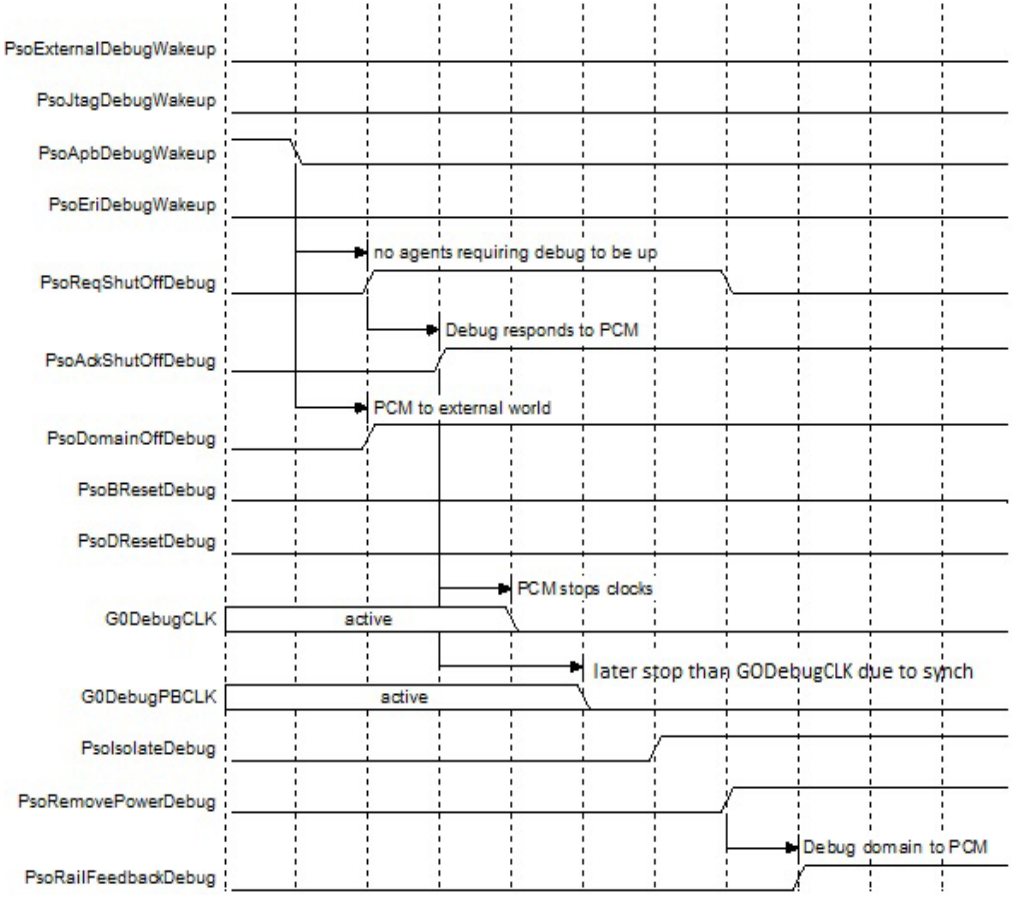


Figure 33–181. No-retention Configuration Debug Domain Shutoff

The retention configuration processor domain shutoff is shown in Figure 33–182. The debug and memory domains do not have retention flops, thus shutoff in the retention configuration is the same as the no-retention configuration.

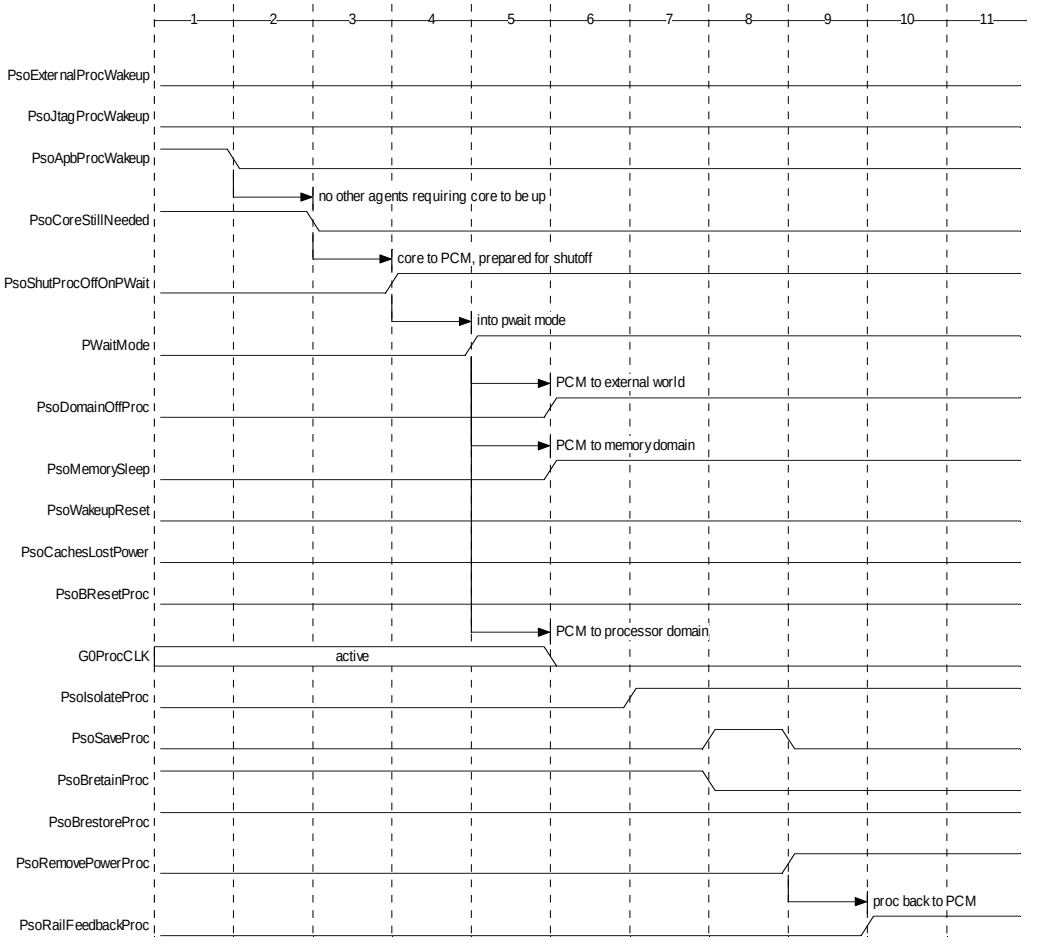


Figure 33–182. Retention Configuration Processor Domain Shutoff

Assuming the Core power domain of the Xtmem subsystem has been in the steady state off condition, a wakeup is initiated by the following:

1. If any one of the Pso*ProcWakeup signals is set logic high, the PCM begins the Core domain wakeup sequence.
2. PsoRemovePowerProc is set logic low to all of the power switches in the Core power domain. At the same time, PsoMemorySleep is also set logic low.

3. The PCM module waits for the feedback signal `PsoRailFeedbackProc` to also go low to ensure that the power switches have all received the `PsoRemovePowerProc` signal.
4. All system clocks (CLK, BCLK) are started and system resets (BReset, DReset) are set logic high for at least six of the slower clock cycles. The resets are then lowered and the clocks are stopped again.
5. (If full state retention is configured) `PsoBretainProc`/`PsoSaveProc`/`PsoBrestoreProc` signals are set to the correct values to restore the state of the state-retention registers. These signals are given two cycles to fan out through the Core power domain.
6. `PsoIsolateProc` is set logic low and given two cycles to fan out through the Core power domain to all of the isolation gates on signal outputs.
7. All system clocks (CLK, BCLK) are permanently restarted.
8. The Xtensa core completes its software wakeup sequence and lowers the PsoShutProcOffOnPWait signal to the PCM module.
9. The `PsoDomainOffProc` signal is lowered to notify the external world that the Core power domain is on.

The no-retention configuration processor domain wakeup is shown in Figure 33–183.

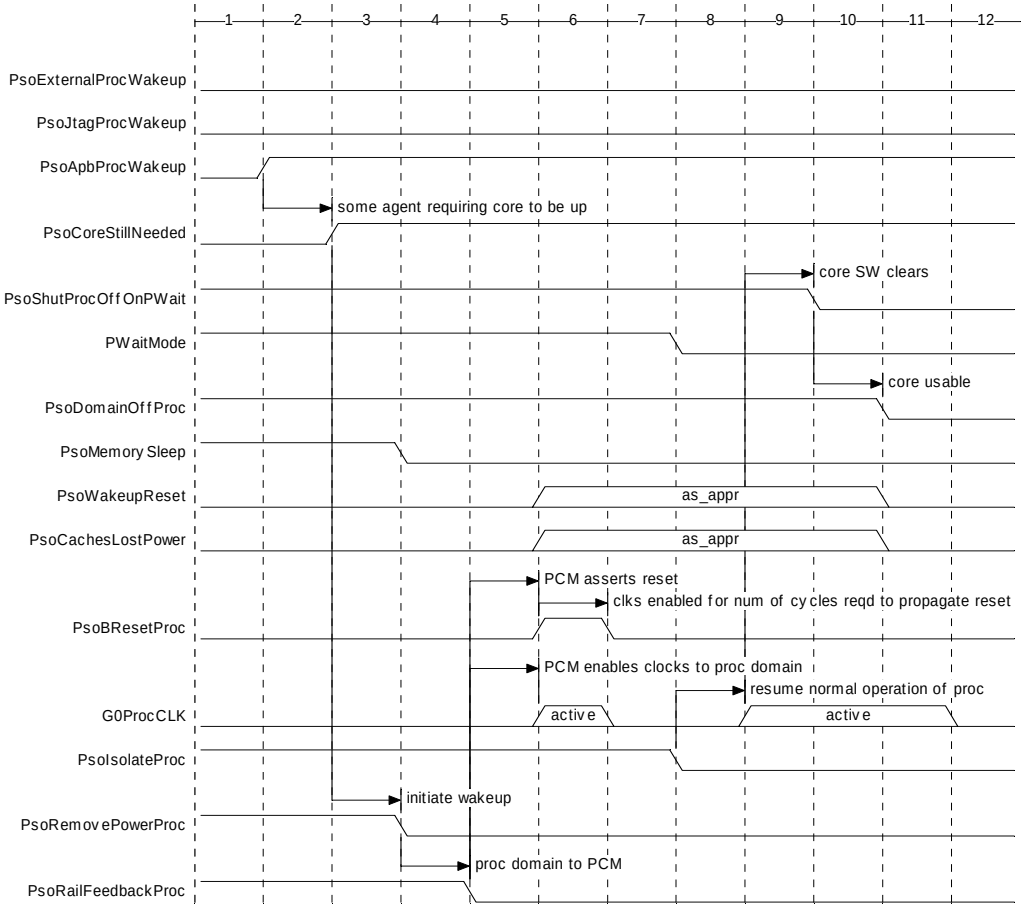


Figure 33–183. No-retention Configuration Processor Domain Wakeup

The no-retention configuration memory domain wakeup is shown in Figure 33–184. The memory domain wakeup sequence is relatively simple and completes in phase 6 with the deassertion of `PsoDomainOffMem`. However, note that the previously-mentioned restriction that the processor domain must be off if the memory domain is off. Processor wakeup subsequent to memory wakeup is therefore a common scenario. Thus, for didactic purposes, Figure 33–184 shows processor domain wakeup also – as it relates to the memory domain. Basically, `PsoMemorySleep` is deasserted as a consequence of the processor wakeup, and reset of the core upon wakeup causes the ERI memory wakeup signal to be preset – to guarantee that the core has its caches and local memories available for access.

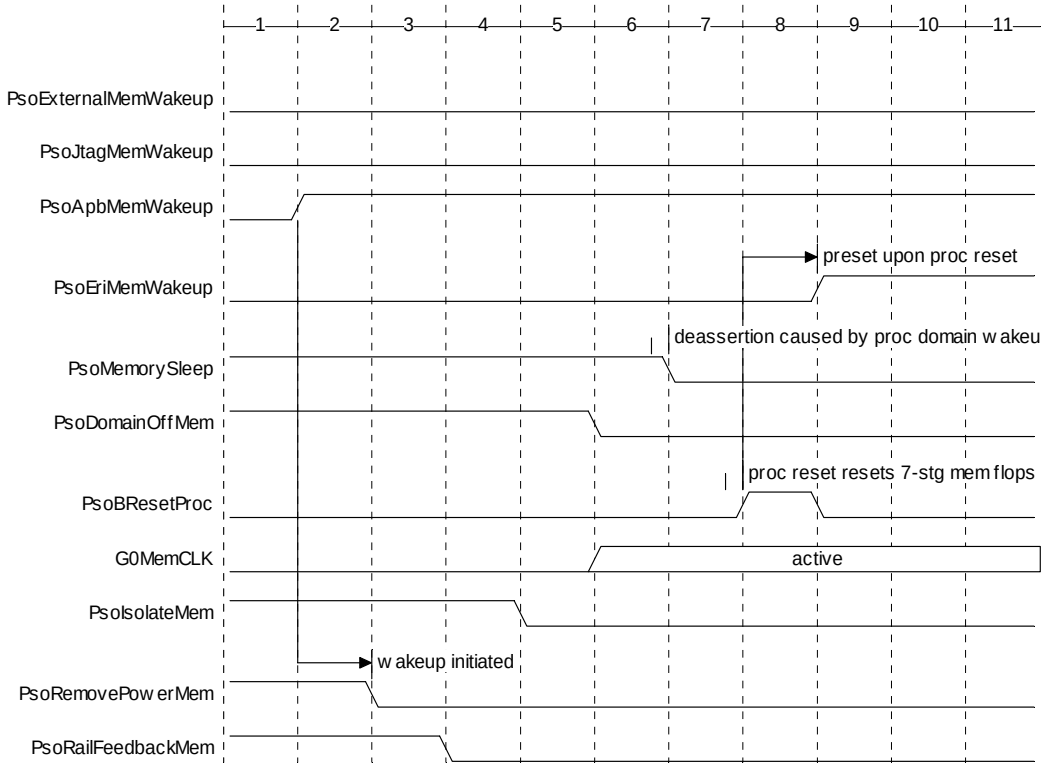


Figure 33–184. No-retention Configuration Memory Domain Wakeup

The no-retention configuration debug domain wakeup is shown in Figure 33–185.

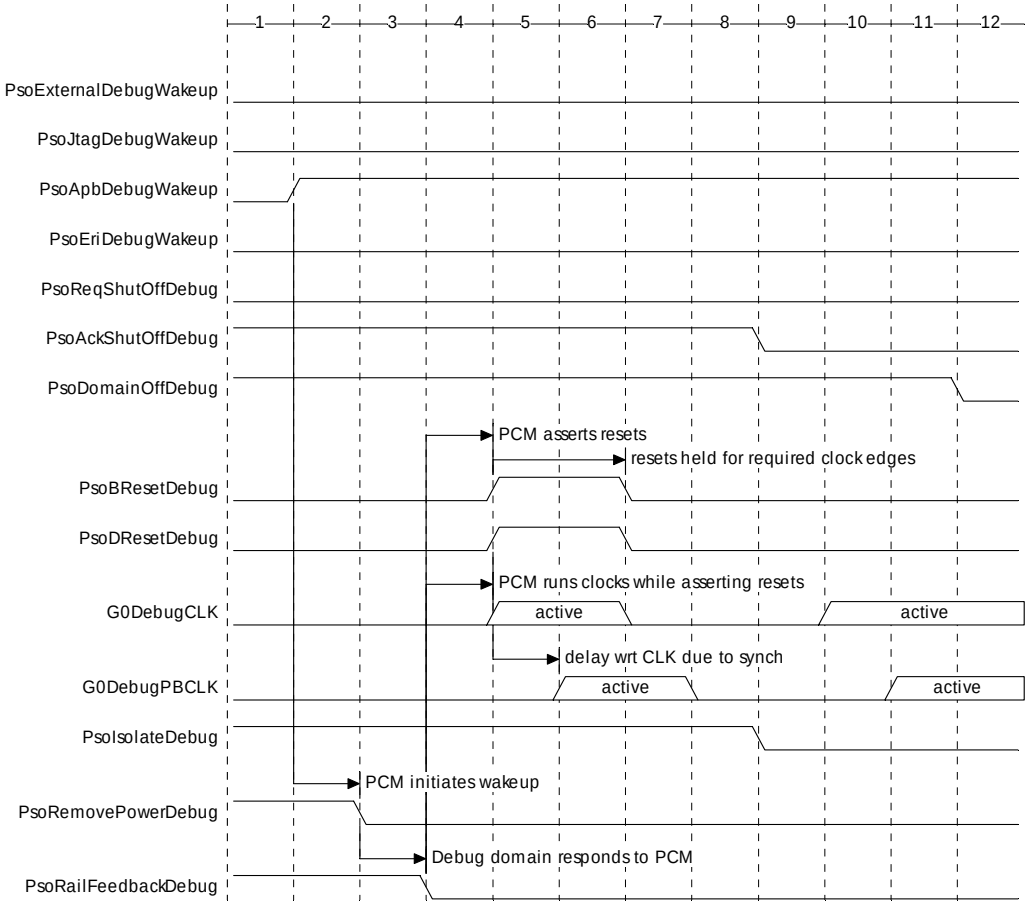


Figure 33–185. No-retention Configuration Debug Domain Wakeup

The retention configuration processor domain wakeup is shown in Figure 33–186. The debug and memory domains do not have retention flops, so wakeup in the retention configuration is the same as the no-retention configuration.

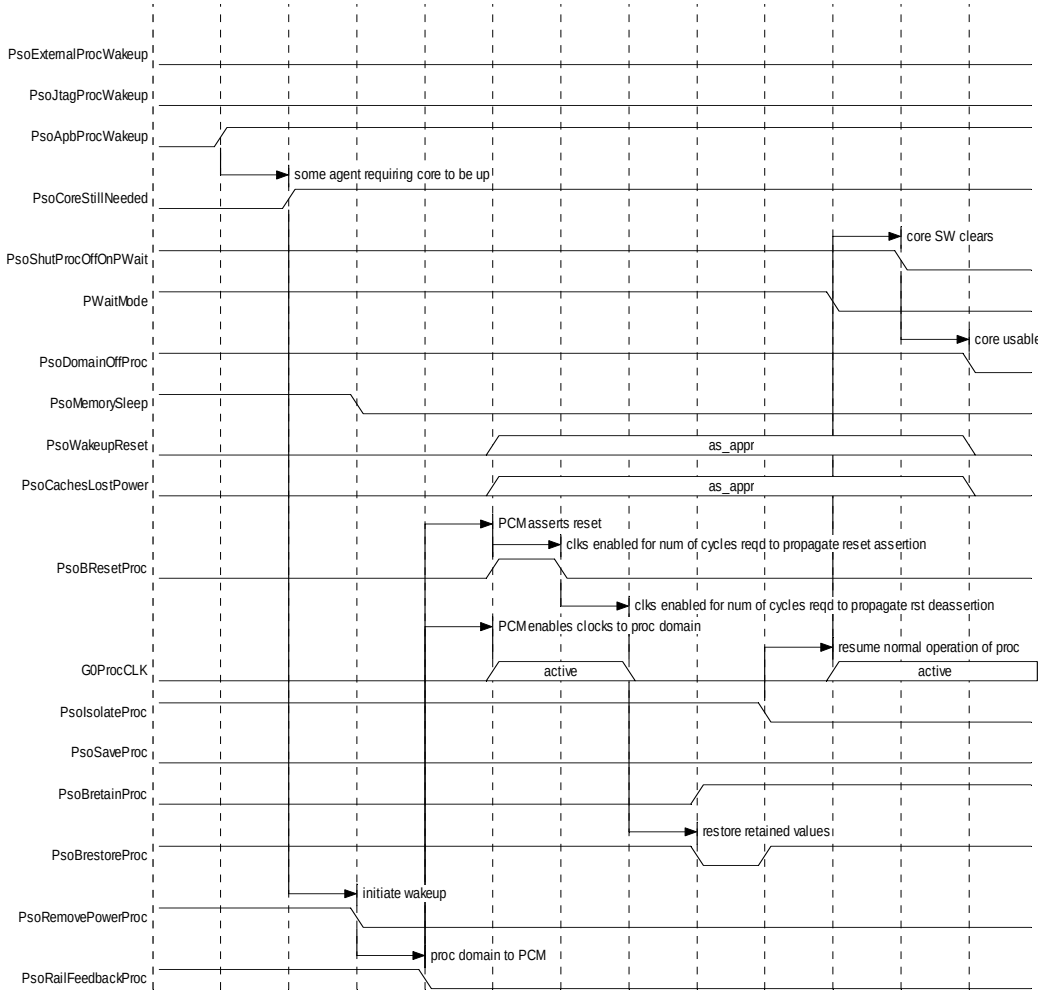


Figure 33–186. Retention Configuration Processor Domain Wakeup

33.5 Reset Behavior of Xtensa with PSO

For configurations with PSO support, the power control module (PCM) normally takes control of the BReset and DReset signals into the processor during both the shutdown and wakeup sequences.[1] It is therefore not necessary for an external agent to assert either BReset or DReset during the wakeup sequence to comply with the 10-cycle Xtensa reset requirement, as this is automatically performed by the PCM. However, both BReset and DReset are ORed with the PCM reset control signal, so if a user wants to keep a core in the reset state during and after the wakeup process, one or both signals can be asserted. The following are example reset cases, where we assume a master processor on the SoC controls the Xtensa subsystem.

Master wants a Full Reset

Regardless of PSO configuration (single-domain / three-domain no-retention / three-domain full-retention), the PcmReset signal should be asserted for at least 10-cycles. This resets both the PCM module and the processor core, and brings the entire system to the full/standard power-on reset state.

Master wants to Wake up Xtensa Core, Followed by Memory Initialization

Regardless of PSO configuration (single-domain / three-domain no-retention / three-domain full-retention), the appropriate Pso*Wakeup signal should be asserted to begin the wakeup sequence. The RunStall signal can then be used to hold the core until the appropriate PsoDomainOff* output signal is driven low, indicating the core is ready for use, at which point the memory initialization sequence can begin.

Master wants to Perform full Reset, Subsequently keep Xtensa Core in Reset

Like the first situation (Master wants a Full Reset) described above, master asserts PcmReset for at least 10-cycles, but also assert the external BReset and DReset signals. This will complete the reset of the PCM module, while still keeping Xtensa in the reset state. This is useful when one core in a multi-core system needs to be synchronized to the others.

1. Other resets such as JTRST and PRESETn are used primarily by flops in the always-ON domain and therefore do not come under control of the PCM. Stated differently, BReset covers all the flops in the Processor domain, and DReset covers all the flops in the Debug domain.

33.5.1 Precautions in using BReset and DReset in PSO Configurations

As stated above, the PCM module controls both BReset and DReset signals during the shutdown and wakeup sequences, so a user should not need to assert these signals. However, since the BReset and DReset signals are ORed with the PCM reset control signal for a user-override capability, the following precautions need to be taken.

1. If the core has been configured for three power domains (core/debug/memory) with full state retention, asserting either BReset or DReset during the wakeup process will prevent the correct operation of the state retention registers. Most vendor libraries will flag this as an error during simulation. It is recommended to wait for the wakeup sequence to complete, and for the PsoDomainOff* output signals to be driven low, before attempting to reset the core again.
2. If a power down request has been initiated, neither BReset nor DReset should be asserted during the power down sequence; this may corrupt core state that the PCM module is depending on to correctly complete the power down sequence. Only after the PsoDomainOff* signals have been driven high, and a user wants to request a core wakeup via the Pso*Wakeup signals, can the BReset or DReset signals be asserted.

34. Xtensa Processor Core Clock Distribution

This chapter describes the clock distribution options and the logic clocking styles within the Xtensa processor core.

34.1 Clock Distribution

A *synchronous* system is one in which all storage elements (such as registers and flip-flops) are clocked simultaneously. The Xtensa processor core is *primarily* a synchronous block, as described in Section 34.2. To ensure correct operation and high performance, a system designer must balance clocks both inside and outside of the processor core.

Figure 34–187 depicts one simple SOC clocking scheme: all clocks are derived from one common source. That clock is then buffered to each block in the chip (such as the processor core, caches and memories, peripherals, and other device logic).

Inside the Xtensa processor core, a *clock-distribution tree* buffers and re-distributes the clock to each flip-flop. The clock-distribution tree may also optionally *gate* the clock, turning off the clock to parts of the core to save power.

34.1.1 Clock Domains within Xtensa

For simplicity, Figure 34–187 shows only one clock input to Xtensa and one tree therein. In truth there are several clock domains in Xtmem, and each has an associated input and tree. The full list of domains are as shown in Figure 34–187:

Table 34–127. Clock Domains

Name	Description	Requirements
CLK	Main clock input to Xtensa - specifically Xtmem. The majority of Xtensa flops run off this clock including the processor core, Debug, bridge logic, etc. Has numerous and multi-level clock gating.	See Note below.
BCLK	Bus clock when core and bus clocks are asynchronous. This clock is gated.	Asynchronous to other clocks. See Note below.

Table 34–127. Clock Domains (continued)

Name	Description	Requirements
JTCK	The standard JTAG clock. Logic in the Debug and Access Port modules that interface to JTAG run off this clock. This clock is never gated.	Asynchronous to other clocks. Must be four or more times slower than CLK.
PBCLK	APB interface clock. The APB slave port logic in Xtensa runs off this clock. This clock is gated. PBCLK is not to be confused with PCLK, the virtual clock described in Section 34.3 “PCLK, the Processor Reference Clock” below.	Asynchronous to other clocks. See Note below.

Note: Except for the clock gating logic that uses latches, the Xtensa design uses all edge-triggered flip-flops. Given that fact, the Xtensa IP does not impose any duty cycle requirements on any of its clocks. The implementation scripts in the hardware package specifies a 50% duty cycle, but you can choose any duty cycle that meets timing in your sign-off flow.

The subsequent sections on this chapter will focus on CLK the main processor clock.

34.1.2 Clocking Xtensa

The processor’s clock tree delays the input clock by an *insertion delay*, shown in Figure 34–187 as *Ti_Xtensa*. This is the delay from the core’s clock input to a leaf node of the clock tree. For all clocks in the SOC to be balanced, they must all have the same total insertion delay from the common clock as the Xtensa processor core.

Chip designers have several options for generating clocks for the rest of the chip. The example of Figure 34–187 assumes that the clock-tree insertion delay for the processor core, *Ti_Xtensa*, is greater than the insertion delay of the other blocks. Here a designer has buffered the clocks connected to the other blocks to balance all the on-chip clocks.

Alternatively, a design could use a single large clock distribution tree to generate a synchronous clock for all points in the system. Again, the same rules about balancing the clocks apply.

Finally, some SOC designs contain multiple clock domains. In this case, all signals that cross into the given Xtensa clock domain must first be synchronized with the Xtensa clock before they connect to the processor input pins. There are some exceptions to this where the signals are assumed to be asynchronous to begin with (for example, the BreakIn signal used in MP debug synchronization). These exceptions are noted in the sections describing these signals.

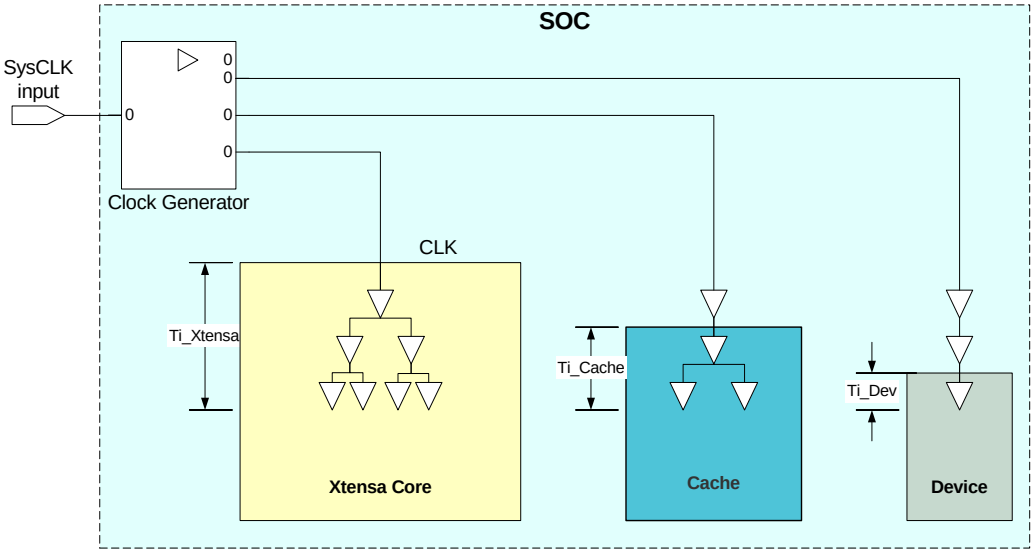


Figure 34–187. Balancing Clock Distribution Trees in an SOC

34.2 Clock Distribution in the Xtensa Processor Core

The following sections discuss the logical and physical treatment of the CLK input clock to the Xtensa processor core.

34.2.1 Clock-Distribution Tree

CLK distribution is illustrated in Figure 34–189. Logically, this clock is used to clock all edge-triggered flip-flops in the core. Electrically, this clock must be fanned out through a clock-distribution tree to drive all devices. To maintain a synchronous system, the main constraint on the clock-distribution tree is that all leaf nodes see the same clock. The *skew* between any leaf node of the Xtensa clock tree should be minimized because clock skew reduces performance.

The scripts supplied with the Xtensa processor RTL allow you to synthesize the core clock-distribution tree. Typically, these scripts constrain the clock skew across any leaf nodes to be 100 psec in 65nm technology.

34.2.2 Clock Gating

The Xtensa clocks allow two levels of clock gating in the clock-distribution tree. The clock-gating option is controlled by two selections. These options are independent of each other and can be selected in any combination. The options are:

- Global Clock Gating — Reduces the processor core's power consumption by using global clock gating during sleep mode.
- Functional Clock Gating — Reduces the processor core's power consumption by using functional clock gating within modules.

Xtensa uses a standard clock-gating circuit to enable and disable clocks cleanly without errors, as shown in Figure 34–188. The clock-gating circuit uses a latch with a low-true enable to control glitching of the clock enable during the high phase of the clock. Many standard cell libraries contain clock-gating cells that either map to the xtgated_tmode_clock or xtgated_clock circuit. Two levels of clock-gating hierarchy are offered to allow better mapping of the clock-gating circuit based on the libraries treatment of the `TModeClkGateOverride` signal.

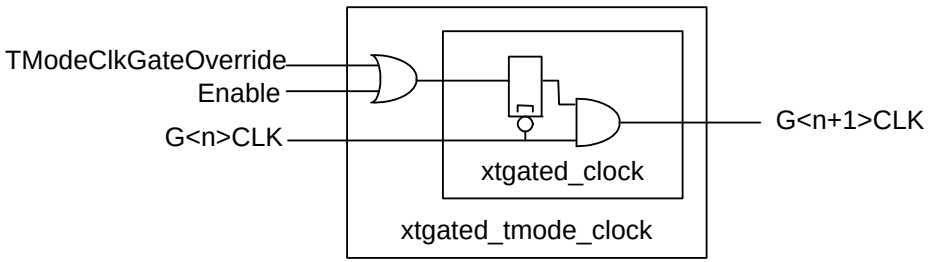


Figure 34–188. Clock Gating Circuit

Implementation of clock gating imposes a maximum insertion-delay restriction on portions of the processor's clock-tree circuitry.

34.2.3 Global Clock Gating for Power

Global clock gating is the Xtensa processor's first clock-gating level. There are three main global clock-gating branches:

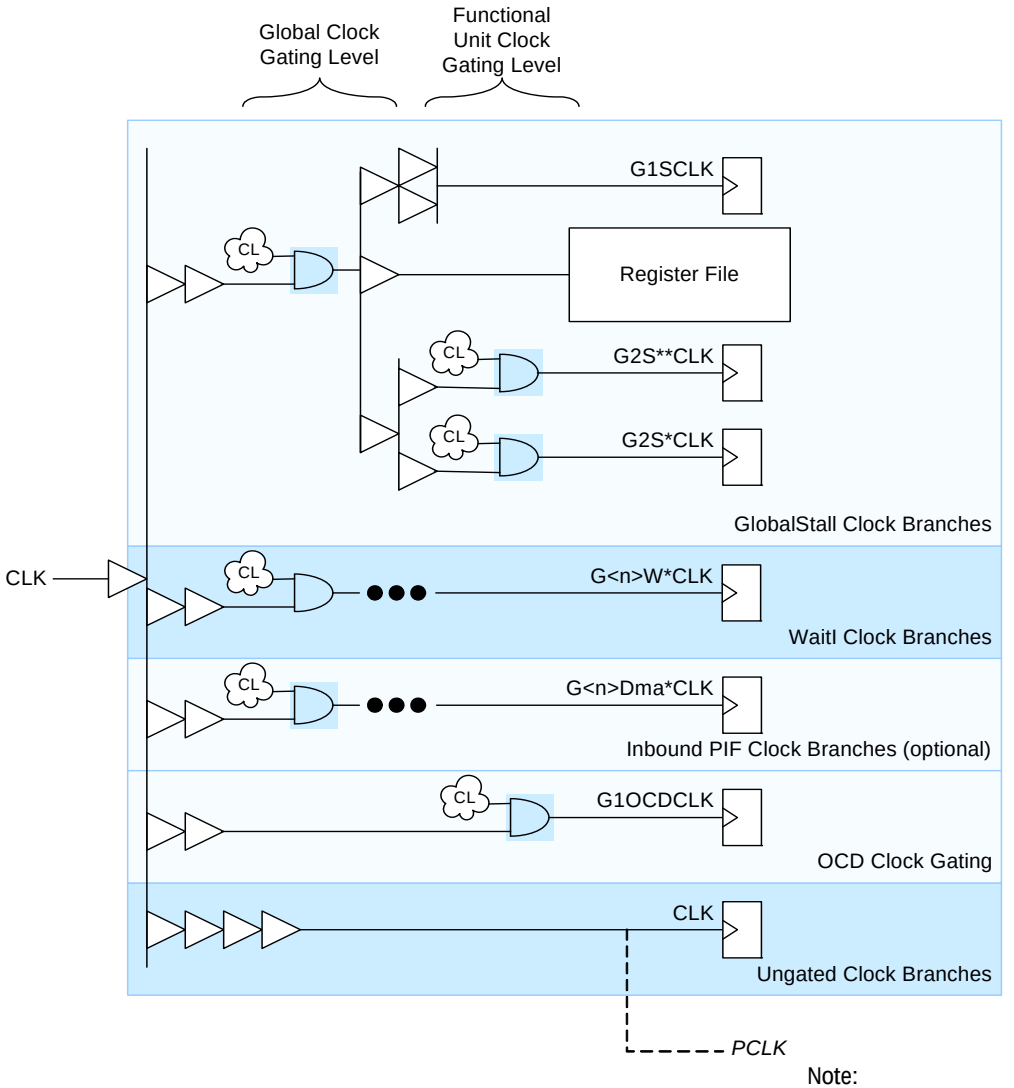
- WAITI clock gating allows the processor to be placed in a low-power mode under program control. When the Xtensa processor executes a WAITI instruction, the processor sets the interrupt level, powers down the clocks to most of the core, and waits for an interrupt. When an interrupt occurs, the processor awakes and re-enables the clock.
- GlobalStall clock gating is used to stall the processor pipeline execution under hardware control. GlobalStall's are generally used to resolve hardware conflicts due to busy memory interfaces. The GlobalStall clock branch is also gated by the WAITI condition. See Appendix A.9 for the list of GlobalStall causes.
- The Inbound-PIF clock branch is generally turned off during a WAITI low power mode, but will temporarily turn on the clocks needed for inbound PIF operations. Inbound PIF operations during a WAITI low power mode may have an extra cycle of latency, which is required to turn on this clock branch.

34.2.4 Functional Clock Gating

If the functional clock-gating configuration option is selected, a second clock-gating level dynamically turns off clocks to different modules within the processor core when those modules are not in use. The clocks are enabled and disabled automatically by internal processor hardware, not under program control.

If *global clock gating* is not selected, then functional clock gating will be performed at the first clock-gating level.

Figure 34–189 illustrates the clock distribution in an Xtensa processor core with two clock-gating levels.



Note:

Internal reference PCLK represents a leaf node of the non-gated clock-distribution tree within the core. All gated and non-gated clocks within the core must be synchronous with PCLK. All Xtensa processor timing diagrams and AC timing specifications are with respect to PCLK.

Figure 34–189. Clock Distribution in an Xtensa Processor Core

34.2.5 Register Files

The Xtensa processor register files are built with flip-flops and are close to a latch-based implementation in power consumption when functional clock gating is selected.

34.3 PCLK, the Processor Reference Clock

Figure 34–189 shows CLK, the clock input to the Xtensa processor core. The figure also labels a clock signal within the processor core named PCLK. This is not an actual signal, but it represents a typical leaf node of the non-gated clock-distribution tree within the processor core. All CLK clocks to registers and flip-flops, whether gated or non-gated, must be synchronous with PCLK.

PCLK is not an output of the core. It is only a timing reference, a “virtual signal.” There is no way to directly observe the Xtensa core clock from outside of the processor core. Note that PCLK is also not to be confused with PBCLK, which is the APB clock.

Cadence provides clock-tree generator scripts for the Xtensa processor core that report both the insertion delay from input CLK to PCLK and the skew between leaf-node clocks. Designers may use these values to balance other clocks on the chip with the Xtensa processor clock.

The actual clock-tree insertion delay for a particular Xtensa processor configuration is not known until after design synthesis, gate placement, and clock-tree generation. For this reason, timing with respect to CLK cannot be specified. However, all clocks within the Xtensa processor core are synchronous with PCLK.

Note: All processor timing diagrams and AC signal times are specified with respect to PCLK. Although PCLK does not exist as a physical signal, it is used as a timing reference throughout this document. Figure 34–190 illustrates how signal set-up and hold times are specified with respect to PCLK.

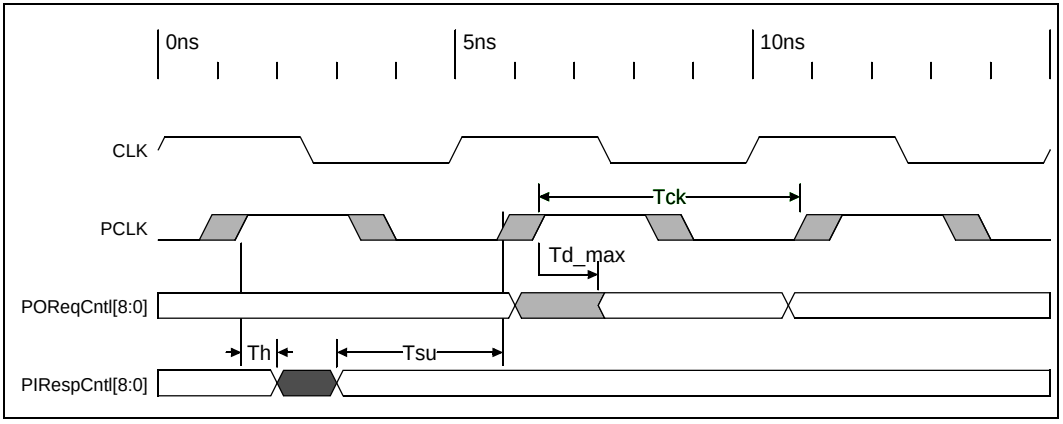


Figure 34–190. Example CLK, PCLK, and Signal Delays for the Xtensa Processor Core

35. Xtensa LX7 Processor AC Timing

The following AC timing specifications help the system designer integrate the synthesizable Xtensa processor core into a system. The AC timing parameters are defined with respect to waveforms shown in Figure 35–191. `CLK` is the clock signal arriving at the processor core's clock port. The clock period is Tck. The PCLK clock signal is a virtual clock reference obtained by adding clock-tree insertion delay to the clock signal `CLK`. The shaded area in the PCLK waveform represents clock jitter and uncertainty (Tunc), which is equal to the difference between maximum clock-tree insertion delay and minimum clock tree insertion delay. `POReqCntl[8:0]` is a representative output signal, and `PIRespCntl[8:0]` is a representative input signal.

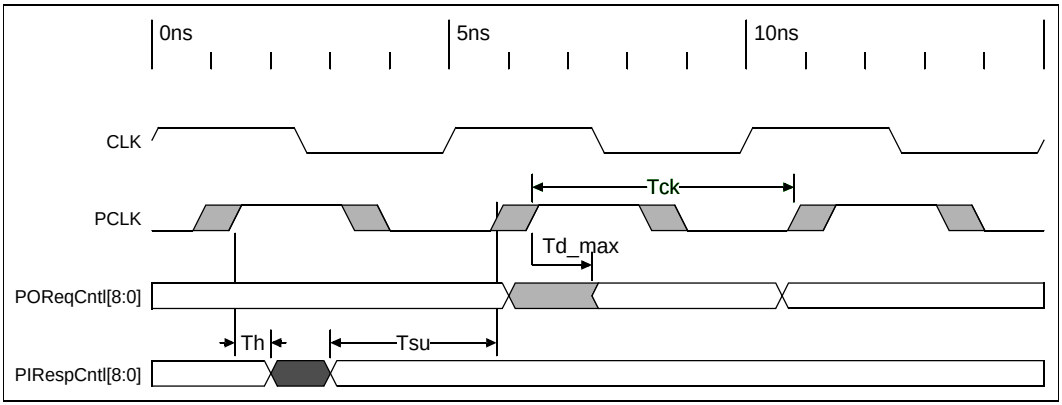


Figure 35–191. CLK, PCLK, Set-up/Hold Time and Output Delays for Processor Core

The AC timing parameters defined for the Xtensa processor core are as follows: the output delay time (Td_max), the input setup time (Tsu), and the input hold time (Th). The output delay time (Td_max) is equal to the maximum delay between the rising edge of PCLK (assuming maximum clock-tree insertion delay) and the time when a signal is available at the output port. In other words, it represents the maximum delay for a signal to propagate through logic that interfaces to an output port.

The input setup time (Tsu) is equal to the minimum time between the time when a signal is required at an input port and the rising edge of PCLK (assuming minimum clock-tree insertion delay). Alternatively, it is the maximum delay for a signal to propagate through logic that interfaces to an input port.

The input hold time (Th) is the minimum time from the rising edge of PCLK assuming maximum clock tree insertion delay to the time when the signal at an input port is allowed to change. The hold time (Th) requirement depends on the flip-flop in the

standard cell library and the minimum timing path in the logic interfacing to an input port. The designer must rely on EDA tools and methodology to ensure that hold times are fixed.

Figure 35–192 shows an example diagram with a PIF output signal, POReqCntl, and a PIF input signal PIRespCntl. The diagram shows how the delay along each PIF signal is budgeted. The Td_max for POReqCntl is 20% of the clock period and Tsu for PIRespCntl is 30% of the clock period.

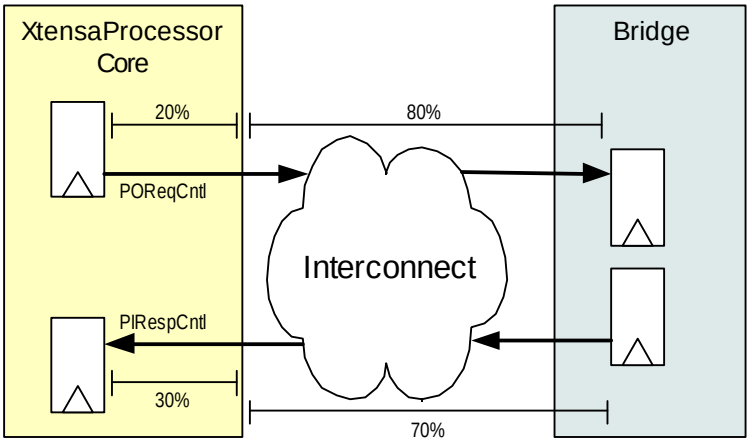


Figure 35–192. PIF Bridge Block Diagram

Note that the delay values specified are time required within the processor core. These values are complementary to the input delay (Tin) and the output delay (Tout) specifications used by EDA tools. The relation between delay specifications used by EDA tools and Xtensa processor AC timing parameters is as follows:

Tin = Tck - Tsu

Tout = Tck - Td_max

It is important to note that the notion of AC timing specifications is only relevant for hard processor cores that run at a specific clock frequency. For synthesizable processor cores, timing-budget flexibility exists between the processor IP and the logic that interfaces to the processor. This flexibility must be exploited fully to optimize an implementation for speed and area.

Tensilica recognizes the designer's need to start implementation with a minimal amount of effort spent on deriving timing constraints, and the competing need to have flexibility in modifying constraints to suit overall system-level time budgeting.

To meet these requirements, the Xtensa processor has a default set of constraints (Tin and Tout) specified in a simple text file, which can be modified to suit specific design requirements. These timing constraints are for timing budgeting purpose. Default constraints are provided for local memory ports, PIF ports, and TIE ports.The constraints for a local memory port depends on the pipeline configuration (5- or 7-stage) and the timing requirements generated by industry-standard memory compilers. The constraints for the PIF port and TIE ports are specified as a percentage of the target clock period.

35.1 Local Memory Timing

The default local memory (cache, RAM, ROM, XLMI) timing constraints for a 5-stage pipeline configuration are based on the access time and the setup time requirements for memories generated by industry-standard memory compilers. The timing requirements for memories are guard banded by an additional 10% to model top-level wiring delays. For example, the default timing constraints based on an industry-standard memory compiler for 65nm TSMC GP process technology are shown in Table 35–128. The table is divided into three parts depending on the number of bits in the local memory interface. Each part specifies the size of the local memory, the address setup time (Tac), the write-enable setup time (Twc), the write data setup time (Tdc), and read data access time (Tcq). The table shows that the (address, write-enable, and write data) setup time increases with an increase in the local memory interface width and also increases with an increase in the memory size. Moreover, the increase is in a narrow range. However, the memory access time increases quite rapidly for memories 16KB or larger. For large memories, the large access time requirement leads to degradation in the system performance as timing paths in the processor that load/fetch the data from the local memory become timing critical.

Table 35–128. Default Timing Constraints for 28nm HPM

28nm HPM High-Speed Memory Data						
32-bit Local Memory Interface (size in bytes)						
Size	**Tac**	**Tmc**	**Twmc**	**Twc**	**Tdc**	**Tcq**
512	0.05	0.10	0.07	0.05	0.07	0.38
1024	0.05	0.10	0.07	0.05	0.07	0.40
2048	0.05	0.10	0.06	0.06	0.06	0.44
4096	0.07	0.10	0.06	0.06	0.06	0.47
8192	0.08	0.10	0.06	0.07	0.06	0.51
16384	0.10	0.10	0.07	0.09	0.07	0.56
32768	0.15	0.10	0.10	0.11	0.10	0.63
65536	0.15	0.10	0.11	0.12	0.11	0.75
64-bit Local Memory Interface (size in bytes)						
Size	**Tac**	**Tmc**	**Twmc**	**Twc**	**Tdc**	**Tcq**
512	0.05	0.10	0.06	0.05	0.06	0.41
1024	0.05	0.10	0.06	0.05	0.06	0.41
2048	0.05	0.10	0.06	0.05	0.06	0.44
4096	0.06	0.10	0.06	0.06	0.06	0.49
8192	0.07	0.10	0.06	0.06	0.06	0.51
16384	0.10	0.10	0.06	0.08	0.06	0.55
32768	0.15	0.10	0.09	0.10	0.09	0.63
65536	0.15	0.10	0.09	0.12	0.09	0.74
131072	0.15	0.10	0.11	0.09	0.11	0.91
128-bit Local Memory Interface (size in bytes)						
Size	**Tac**	**Tmc**	**Twmc**	**Twc**	**Tdc**	**Tcq**
512	0.05	0.10	0.05	0.06	0.05	0.52
1024	0.05	0.10	0.05	0.06	0.05	0.52
2048	0.05	0.10	0.05	0.06	0.05	0.52
4096	0.05	0.10	0.05	0.06	0.05	0.56

Notes:

Tac = address-to-clock setup time

Tmc =memory-enable-to-clock setup time

Twmc = bit-write-enable-to-clock setup time

Twc = write-enable-to-clock setup time

Tdc = write-data-to-clock setup time

Tcq = clock-to-q (data out) delay time

The memory times are scaled by 1.1 to allow 10% budget for the interconnect delay

Table 35–128. Default Timing Constraints for 28nm HPM (continued)

28nm HPM High-Speed Memory Data						
128-bit Local Memory Interface (size in bytes) (continued)						
Size	**Tac**	**Tmc**	**Twmc**	**Twc**	**Tdc**	**Tcq**
8192	0.06	0.10	0.05	0.06	0.05	0.60
16384	0.08	0.10	0.05	0.07	0.05	0.61
32768	0.10	0.10	0.06	0.09	0.06	0.66
65536	0.15	0.10	0.08	0.11	0.08	0.73
131072	0.15	0.10	0.09	0.11	0.09	0.90

Notes:

Tac = address-to-clock setup time

Tmc =memory-enable-to-clock setup time

Twmc = bit-write-enable-to-clock setup time

Twc = write-enable-to-clock setup time

Tdc = write-data-to-clock setup time

Tcq = clock-to-q (data out) delay time

The memory times are scaled by 1.1 to allow 10% budget for the interconnect delay

Table 35–129. Default Timing Constraints for 40nm LP

40nm LP High-Density Memory Data						
32-bit Local Memory Interface (size in bytes)						
Size	**Tac**	**Tmc**	**Twmc**	**Twc**	**Tdc**	**Tcq**
512	0.24	0.32	0.14	0.29	0.14	1.08
1024	0.24	0.32	0.13	0.29	0.13	1.13
2048	0.24	0.33	0.13	0.33	0.13	1.19
4096	0.25	0.33	0.16	0.32	0.16	1.36
8192	0.25	0.33	0.13	0.37	0.13	1.44
16384	0.26	0.33	0.16	0.39	0.16	1.54
32768	0.33	0.33	0.24	0.46	0.24	1.64
65536	0.30	0.33	0.21	0.52	0.21	1.88
64-bit Local Memory Interface (size in bytes)						
Size	**Tac**	**Tmc**	**Twmc**	**Twc**	**Tdc**	**Tcq**
512	0.25	0.33	0.09	0.29	0.09	1.08
1024	0.25	0.33	0.09	0.29	0.09	1.11
2048	0.24	0.33	0.09	0.29	0.09	1.17
4096	0.24	0.33	0.05	0.30	0.05	1.36
8192	0.25	0.32	0.13	0.32	0.13	1.41
16384	0.24	0.33	0.16	0.35	0.16	1.50
32768	0.29	0.32	0.22	0.41	0.22	1.61
65536	0.34	0.33	0.13	0.43	0.13	1.84
131072	0.31	0.34	0.08	0.48	0.08	2.23
128-bit Local Memory Interface (size in bytes)						
Size	**Tac**	**Tmc**	**Twmc**	**Twc**	**Tdc**	**Tcq**
512	0.25	0.33	0.01	0.29	0.01	1.23
1024	0.25	0.33	0.01	0.29	0.01	1.23
2048	0.25	0.33	0.01	0.29	0.01	1.27
4096	0.24	0.33	0.01	0.30	0.01	1.33

Notes:

Tac = address-to-clock setup time

Tmc =memory-enable-to-clock setup time

Twmc = bit-write-enable-to-clock setup time

Twc = write-enable-to-clock setup time

Tdc = write-data-to-clock setup time

Tcq = clock-to-q (data out) delay time

The memory times are scaled by 1.1 to allow 10% budget for the interconnect delay

Table 35–129. Default Timing Constraints for 40nm LP (continued)

40nm LP High-Density Memory Data						
128-bit Local Memory Interface (size in bytes) (continued)						
Size	**Tac**	**Tmc**	**Twmc**	**Twc**	**Tdc**	**Tcq**
8192	0.25	0.33	0.01	0.30	0.01	1.58
16384	0.25	0.33	0.06	0.32	0.06	1.61
32768	0.24	0.33	0.08	0.36	0.08	1.71
65536	0.29	0.33	0.15	0.41	0.15	1.81
131072	0.41	0.34	0.04	0.42	0.04	2.17

Notes:
Tac = address-to-clock setup time
Tmc =memory-enable-to-clock setup time
Twmc = bit-write-enable-to-clock setup time
Twc = write-enable-to-clock setup time
Tdc = write-data-to-clock setup time
Tcq = clock-to-q (data out) delay time
The memory times are scaled by 1.1 to allow 10% budget for the interconnect delay

Use of the 7-stage Xtensa LX7 pipeline configuration reduces the problem of slower memory access times associated with larger local memories. The 7-stage configuration inserts an extra clock cycle for the read response from the local memory. The memory data can be registered by a flip-flop. The default input delay memory constraints (Tin) for a 7-stage Xtensa LX7 pipeline configuration are equal to a flip-flop's clock-to-q delay. In Table 35–148 through Table 35–157, the input setup delay AC timing specifications are larger for the 7-stage-pipeline configuration than for the 5-stage-pipeline version because the memories for 7-stage-configurations have an additional clock cycle to make the read data available.

35.2 Processor Interface (PIF) AC Timing

Table 35–130 and Table 35–131 specify the input and output PIF timing required for a typical design based on an Xtensa processor. Tensilica recommends that all devices driving PIF signals should drive them directly from a flip-flop and that minimal logic should be placed on PIF signals before clocking them into an input register. Budgeting the maximum possible time in the interconnect allows for the greatest flexibility in interconnect design and in chip floor planning.

Table 35–130. PIF Output Signal Timing

PIF Output Signal Name	Td_max
POReqID	20% of PCLK
POReqValid	20% of PCLK
POReqCntl	20% of PCLK
POReqAdrs	20% of PCLK
POReqDataBE	20% of PCLK
POReqData	20% of PCLK
POReqPriority	20% of PCLK
POReqAttribute	20% of PCLK
PORespRdy	50% of PCLK
PORespValid	20% of PCLK
PORespCntl	20% of PCLK
PORespData	20% of PCLK
PORespId	20% of PCLK
PORespPriority	20% of PCLK
POReqRdy	20% of PCLK

Table 35–131. PIF Input Signal Timing

PIF Input Signal Name	Set up (Tsu)
PIReqID	30% of PCLK
PIRespRdy	30% of PCLK
PIRespData	30% of PCLK
PIRespCntl	30% of PCLK
PIRespValid	30% of PCLK
PIRespPriority	30% of PCLK
PIRespId	30% of PCLK
PIReqRdy	40% of PCLK
PIReqValid	30% of PCLK
PIReqCntl	30% of PCLK
PIReqAdrs	30% of PCLK
PIReqData	30% of PCLK

Table 35–131. PIF Input Signal Timing (continued)

PIF Input Signal Name	Set up (Tsu)
PIReqDataBE	30% of PCLK
PIReqPriority	30% of PCLK
PIReqAttribute	30% of PCLK

35.2.1 AHB-Lite and AXI Bus Bridge AC Timing

Table 35–132 lists the timing of the clock enable signal, Strobe, which is used to allow the processor to run at an integer multiple of the bus clock.

Table 35–132. Bus Bridge Clock Enable

Signal	Input Setup
Strobe	50%

Table 35–133 and Table 35–134 list the AHB-Lite bus bridge timing specifications. Table 35–135 and Table 35–138 list the AXI bus bridge timing specifications. All AHB-Lite and AXI timing specifications are relative to either the processor or bus clock as follows:

- For configurations with the asynchronous AMBA bus option, all timing specifications are relative to the bus clock, BCLK.
- For configurations without the asynchronous AMBA bus option, all timing specifications are relative to the processor clock, CLK.

Table 35–133. AHB-Lite Master Interface Timing Specification

Signal	Input Setup (Tsu)	Output Delay (Td_max)
HTRANS		30%
HWRITE		30%
HLOCK		30%
HBURST		30%
HSIZE		30%
HPROT		30%
HADDR		30%
HWDATA		30%
HXTUSER		30%

Table 35–133. AHB-Lite Master Interface (continued) **Timing**

Signal	Input Setup (Tsu)	Output Delay (Td_max)
HREADY	40%	
HRESP	40%	
HRDATA	30%	

Table 35–134. AHB-Lite Slave Interface Timing Specification

Signal	Input Set up (Tsu)	Output Delay (Td_max)
HSEL_S	40%	
HTRANS_S	40%	
HWRITE_S	40%	
HMASTLOCK_S	40%	
HBURST_S	40%	
HSIZE_S	40%	
HPROT_S	40%	
HADDR_S	40%	
HWDATA_S	40%	
HREADY_OUT_S		30%
HRESP_S		30%
HRDATA_S		30%

Table 35–135. AXI Master Interface Timing Specification

Signal	Input Setup (Tsu)	Output Delay (Td_max)
ARADDR		30%
ARBURST		30%
ARCACHE		30%
ARID		30%
ARLEN		30%
ARLOCK		30%
ARPROT		30%
ARQOS		30%
ARXTUSER		30%
ARREADY	50%	
ARSIZE		30%
ARVALID		30%
AWADDR		30%
AWBURST		30%
AWCACHE		30%
AWID		30%
AWLEN		30%
AWLOCK		30%
AWPROT		30%
AWQOS		30%
AWXTUSER		30%
AWREADY	50%	
AWSIZE		30%
AWVALID		30%
BID	30%	
BREADY		30%
BRESP	30%	
BVALID	50%	
RDATA	30%	
RID	30%	
RLAST	30%	
RREADY		30%

Table 35–135. AXI Master Interface Timing Specification (continued)

Signal	Input Setup (Tsu)	Output Delay (Td_max)
RRESP	30%	
RVALID	50%	
WDATA		30%
WID		30%
WLAST		30%
WREADY	50%	
WSTRB		30%
WVALID		30%
WREADY	50%	
WSTRB		30%
WVALID		30%
ARSNOOP		30%
ARBAR		30%
ARDOMAIN		30%
AWSNOOP		30%
AWBAR		30%
AWDOMAIN		30%
AXISECMST	30%	
ARADDRPTY		30%
ARCNTLPTY		30%
ARVPTY		30%
ARRPTY	30%	
RRPTY		30%
RCNTLPTY	40%	
RDATAECC	40%	
RVPTY	40%	
AWADDRPTY		30%
AWCNTLPTY		30%
AWVPTY		30%
AWRPTY	30%	
WCNTLPTY		30%
WDATAECC	30%	

Table 35–135. AXI Master Interface Timing Specification (continued)

Signal	Input Setup (Tsu)	Output Delay (Td_max)
WSTRBPTY		30%
WVPTY		30%
WRPTY	30%	
BRPTY		30%
BCNTLPTY	30%	
BVPTY	30%	
AXIPARITYSEL	30%	

Table 35–136. AXI iDMA Master Interface Timing Specification

Signal	Input Set up (Tsu)	Output Delay (Td_max)
ARADDR_iDMA		30%
ARBURST_iDMA		30%
ARCACHE_iDMA		30%
ARID_iDMA		30%
ARLEN_iDMA		30%
ARLOCK_iDMA		30%
ARPROT_iDMA		30%
ARQOS_iDMA		30%
ARXTUSER_iDMA		30%
ARREADY_iDMA	50%	
ARSIZE_iDMA		30%
ARVALID_iDMA		30%
AWADDR_iDMA		30%
AWBURST_iDMA		30%
AWCACHE_iDMA		30%
AWID_iDMA		30%
AWLEN_iDMA		30%
AWLOCK_iDMA		30%
AWPROT_iDMA		30%
AWQOS_iDMA		30%
AWXTUSER_iDMA		30%

Table 35–136. AXI iDMA Master Interface Timing Specification

Signal	Input Set up (Tsu)	Output Delay (Td_max)
AWREADY_iDMA	50%	
AWSIZE_iDMA		30%
AWVALID_iDMA		30%
BID_iDMA	30%	
BREADY_iDMA		30%
BRESP_iDMA	30%	
BVALID_iDMA	50%	
RDATA_iDMA	30%	
RID_iDMA	30%	
RLAST_iDMA	30%	
RREADY_iDMA		30%
RRESP_iDMA	30%	
RVALID_iDMA	50%	
WDATA_iDMA		30%
WID_iDMA		30%
WLAST_iDMA		30%
WREADY_iDMA	50%	
WSTRB_iDMA		30%
WVALID_iDMA		30%
WREADY_iDMA	50%	
WSTRB_iDMA		30%
WVALID_iDMA		30%
ARSNOOP_iDMA		30%
ARBAR_iDMA		30%
ARDOMAIN_iDMA		30%
AWSNOOP_iDMA		30%
AWBAR_iDMA		30%
AWDOMAIN_iDMA		30%
ARADDRPTY_iDMA		30%
ARCNTLPTY_iDMA		30%
ARVPTY_iDMA		30%
ARRPTY_iDMA	30%	
RRPTY_iDMA		30%

Table 35–136. AXI iDMA Master Interface Timing Specification

Signal	Input Set up (Tsu)	Output Delay (Td_max)
RCNTLPTY_iDMA	40%	
RDATAECC_iDMA	40%	
RVPTY_iDMA	40%	
AWADDRPTY_iDMA		30%
AWCNTLPTY_iDMA		30%
AWVPTY_iDMA		30%
AWRPTY_iDMA	30%	
WCNTLPTY_iDMA		30%
WDATAECC_iDMA	30%	
WSTRBPTY_iDMA		30%
WVPTY_iDMA		30%
WRPTY_iDMA	30%	
BRPTY_iDMA		30%
BCNTLPTY_iDMA	30%	
BVPTY_iDMA	30%	

Table 35–137. AXI iDMA Master Interface Timing Specification

AWDOMAIN_iDMA		30%
ARADDRPTY_iDMA		30%
ARCNTLPTY_iDMA		30%
ARVPTY_iDMA		30%
ARRPTY_iDMA	30%	
RRPTY_iDMA		30%
RCNTLPTY_iDMA	40%	
RDATAECC_iDMA	40%	
RVPTY_iDMA	40%	
AWADDRPTY_iDMA		30%
AWCNTLPTY_iDMA		30%
AWVPTY_iDMA		30%
AWRPTY_iDMA	30%	

Table 35–137. AXI iDMA Master Interface Timing Specification

WCNTLPTY_iDMA		30%
WDATAECC_iDMA	30%	
WSTRBPTY_iDMA		30%
WVPTY_iDMA		30%
WRPTY_iDMA	30%	
BRPTY_iDMA		30%
BCNTLPTY_iDMA	30%	
BVPTY_iDMA	30%	

Table 35–138. AXI Slave Interface Timing Specification

Signal	Input Setup (Tsu)	Output Delay (Td_max)
ARADDR_S	30%	
ARBURST_S	30%	
ARCACHE_S	30%	
ARID_S	30%	
ARLEN_S	30%	
ARLOCK_S	30%	
ARPROT_S	30%	
ARQOS_S	30%	
ARREADY_S		30%
ARSIZE_S	30%	
ARVALID_S	30%	
AWADDR_S	30%	
AWBURST_S	30%	
AWCACHE_S	30%	
AWID_S	30%	
AWLEN_S	30%	
AWLOCK_S	30%	
AWPROT_S	30%	
AWQOS_S	30%	
AWREADY_S		30%
AWSIZE_S	30%	
AWVALID_S	30%	

Table 35–138. AXI Slave Interface Timing Specification (continued)

Signal	Input Setup (Tsu)	Output Delay (Td_max)
BID_S		30%
BREADY_S	50%	
BRESP_S		30%
BVALID_S		30%
RDATA_S		30%
RID_S		30%
RLAST_S		30%
RREADY_S	50%	
RRESP_S		30%
RVALID_S		30%
WDATA_S	30%	
WID_S	30%	
WLAST_S	30%	
WREADY_S		30%
WSTRB_S	30%	
WVALID_S	30%	
AXISECSLV	30%	
ARADDRPTY_S	30%	
ARCNTLPTY_S	30%	
ARVPTY_S	30%	
ARRPTY_S		30%
RRPTY_S	30%	
RCNTLPTY_S		30%
RDATAECC_S		30%
RVPTY_S		30%
AWADDRPTY_S	30%	
AWCNTLPTY_S	30%	
AWVPTY_S	30%	
AWRPTY_S		30%
WCNTLPTY_S	30%	
WDATAECC_S		30%

Table 35–138. AXI Slave Interface Timing Specification (continued)

Signal	Input Setup (Tsu)	Output Delay (Td_max)
WSTRBPTY_S	30%	
WVPTY_S	30%	
WRPTY_S		30%
BRPTY_S	30%	
BCNTLPTY_S		30%
BVPTY_S		30%

35.3 Processor Control and Status AC Timing

Table 35–139 lists the set-up timing for the Xtensa processor's control input signals.

Table 35–139. Processor Control Input Signal Timing

Input Signal Name	Set up (Tsu)
BReset	75% of PCLK
PRID	75% of PCLK
RunStall	20% of PCLK
StatVectorSel	75% of PCLK
BInterrupt	60% of PCLK

The processor's reset signal, BReset is synchronized within the processor. See Section 22.3 "Reset" for more information about the processor reset.

Note that the StatVectorSel and PRID ports are not a dynamic inputs—they are normally expected to be hardwired inputs. The StatVectorSel and PRID port inputs must be stable and constant for at least 10 cycles before and 10 cycles after BReset is deasserted. See Section 22.4 "Static Vector Select" for more information.

Table 35–140 lists the timing for the Xtensa processor's status output signals.

Table 35–140. Processor Status Output Signal Timing

Output Signal Name	Td_max
PWaitMode	15% of PCLK
PReset	15% of PCLK
ErrorCorrected	15% of PCLK

35.4 Debug Signal AC Timing

35.4.1 Test and OCD Signals

The following table lists the Test and OCD signals' timing specification.

Table 35–141. Test and OCD Signal Timing Specification

Name	Input Setup (Tsu)	Output Delay (Td_max)
DReset	75% of PCLK	
OCDHaltOnReset	30% of PCLK	
XOCDMode		30% of PCLK
DebugMode		30% of PCLK
TMode	no requirement	
TModeClkGateOverride	no requirement	
BreakIn	asynchronous	
BreakInAck		30% of PCLK
BreakOut		30% of PCLK
BreakOutAck	asynchronous	
DBGEN	asynchronous	
NIDEN	asynchronous	
SPIDEN	asynchronous	
SPNIDEN	asynchronous	

35.4.2 Traceport Signals

Table 35–142 lists the Traceport signals' timing specification.

Table 35–142. Traceport Timing Specification

Name	Input Setup (Tsu)	Output Delay (Td_max)
PDebugEnable	15% of PCLK	
PDebugInst		20% of PCLK
PDebugStatus		20% of PCLK
PDebugData		20% of PCLK
PDebugPC		20% of PCLK
PDebugLS{0,1,2}Stat		20% of PCLK
PDebugLS{0,1,2}Addr		20% of PCLK
PDebugLS{0,1,2}Data		20% of PCLK
PDebugOutPif		20% of PCLK
PDebugInbPif		20% of PCLK
PDebugPrefetchLookup		20% of PCLK
PDebugPrefetchL1Fill		20% of PCLK
PDebugiDMA		20% of PCLK

35.4.3 TRAX AC Timing

Table 35–143 lists the TRAX port timing specification.

Table 35–143. TRAX Timing Specification

Name	Input Setup (Tsu)	Output Delay (Td_max)
TraceMemAddr		60% of PCLK
TraceMemData	30% of PCLK	
TraceMemEn		60% of PCLK
TraceMemWr		60% of PCLK
TraceMemWrData		60% of PCLK
TraceMemReady	70% of PCLK	
CrossTriggerIn	asynchronous	
CrossTriggerInAck		30% of PCLK
CrossTriggerOut		30% of PCLK
CrossTriggerOutAck	asynchronous	
ATCLK	unused	
ATCLKEN		60% of PCLK

Table 35–143. TRAX Timing Specification (continued)

Name	Input Setup (Tsu)	Output Delay (Td_max)
ATRESETn	75% of PCLK	
ATVALID		60% of PCLK
ATREADY	70% of PCLK	
ATID		30% of PCLK
ATBYTES		40% of PCLK
ATDATA		70% of PCLK
AFVALID	60% of PCLK	
AFREADY		40% of PCLK

35.4.4 JTAG AC Timing

Table 35–144 lists the timing specifications of the JTAG interface ports. These are naturally with respect to JTCK.

Table 35–144. JTAG Timing

Name	Input Setup (Tsu)	Output Delay (Td_max)
JTRST	15% of JTCK	
JTDI	15% of JTCK	
JTMS	15% of JTCK	
JTDO		85% of JTCK
JTDOEn		85% of JTCK

Note that JTCK flops are asynchronous-reset only, so there is no absolute requirement that JTRST has to meet with respect to JTCK. However, synthesis and place & route results are of a better quality if JTRST is not left totally unconstrained. Therefore, the back-end implementation scripts that come with Xtensa processors have the constraint specified on the first row of Table 35–145, but note that you can modify this to suit your reset circumstances.

35.4.5 APB AC Timing

Table 35–145 lists the timing specification of the APB interface ports. These are all naturally with respect to PBCLK.

Table 35–145. APB Timing

Name	Input Setup (Tsu)	Output Delay (Td_max)
PBCLKEN	60% of PBCLK	
PRESETn	75% of PBCLK	
PADDR	70% of PBCLK	
PSEL	70% of PBCLK	
PENABLE	70% of PBCLK	
PREADY		70% of PBCLK
PWRITE	70% of PBCLK	
PWDATA	70% of PBCLK	
PRDATA		70% of PBCLK
PSLVERR		70% of PBCLK

35.5 Power Control Module (PCM) AC Timing

Table 35–146 lists the timing specification of the PCM ports.

Table 35–146. PCM Port Timing

Name	Clock Domain	Input Setup (Tsu)	Output Delay (Td_max)
PsoExternalProcWakeup	CLK	30%	
PsoDomainOffProc	CLK	30%	
PsoShutProcOffOnPWait	CLK	30%	
PsoExternalMemWakeup	CLK	30%	
PsoDomainOffMem	CLK	30%	
PsoExternalDebugWakeup	CLK	30%	
PsoDomainOffDebug	CLK	30%	
PcmReset	CLK	30%	

35.6 Cache Interface (CIF) AC Timing

The following notes refer to subsequent AC timing tables for the cache interfaces.

Note:

The constraints for a local memory port depends on the pipeline configuration (5- or 7-stage) and the timing requirements generated by industry-standard memory compilers.

The minimum clock period must be greater than either:

- the processor output delay (core to RAM) + RAM array set-up time + clock uncertainty.
- the processor input set-up time + RAM output delay time + clock uncertainty.

The designer should use the worst-case clock period for specification purposes.

In the following tables, <*way*> refers to the *way* in a set-associative cache, for example, for a 4-way set associative cache, <*way*> can be either A, B, C, or D.

The <lsu> in the tables refer to the load/store unit that is connected to the local memory and is left blank when the C-Box option is selected. For example, the values of <lsu> for a machine with two load/store units can be 0 and 1 without the C-Box, or it can be B0...B3 depending on whether the memories are banked.

Table 35–147. Cache Interface (CIF) Output Signal Timing

Output Signal Name	Td_max
ICacheAddr	85% of PCLK
ICacheWrData	85% of PCLK
ICacheCheckWrData	85% of PCLK
ICache<way>En	85% of PCLK
ICache<way>WordEn	85% of PCLK
ICache<way>Wr	85% of PCLK
ITagAddr	85% of PCLK
ITagWrData	85% of PCLK
ITagCheckWrData	85% of PCLK
ITag<way>En	85% of PCLK
ITag<way>Wr	85% of PCLK
DCacheAddr<lsu>	85% of PCLK
DCacheWrData<lsu>	85% of PCLK
DCacheCheckWrData<lsu>	85% of PCLK
DCacheByteEn<lsu>	85% of PCLK

Table 35–147. Cache Interface (CIF) Output Signal Timing (continued)

Output Signal Name	Td_max
DCache<way>En<lsu>	85% of PCLK
DCache<way>Wr<lsu>	85% of PCLK
DTagAddr<lsu>	85% of PCLK
DTagWrData<lsu>	85% of PCLK
DTagCheckWrData<lsu>	85% of PCLK
DTag<way>En<lsu>	85% of PCLK
DTag<way>Wr<lsu>	85% of PCLK

Table 35–148. Cache Interface (CIF) Input Signal Timing

Input Signal Name	Set up (5-stage pipeline)	Set up (7-stage pipeline)
DCache<*way*>Data<lsu>	40% of PCLK	85% of PCLK
DCache<*way*>CheckData<lsu>	40% of PCLK	85% of PCLK
DTag<*way*>Data<lsu>	60% of PCLK	85% of PCLK
DTag<*way*>CheckData<lsu>	60% of PCLK	85% of PCLK
ICache<*way*>Data	40% of PCLK	85% of PCLK
ICache<*way*>CheckData	40% of PCLK	85% of PCLK
ITag<*way*>Data	60% of PCLK	85% of PCLK
ITag<*way*>CheckData	60% of PCLK	85% of PCLK

35.7 RAM Interface AC Timing

Note: In the following tables <iid> and <did> refer to the IRam and DRam ID, which can take the value 0 or 1.

<lsu> is the DRam signal name suffix that can be load/store unit number (0/1), or bank ID if multiple banks are configured, or left blank when the C-Box option is selected for single-bank two load/store configurations.

Table 35–149. RAM Interface Output Signal Timing

Output Signal Name	Td_max
DRam<did>Addr<lsu>	85% of PCLK
DRam<did>En<lsu>	85% of PCLK
DRam<did>Wr<lsu>	85% of PCLK
DRam<did>ByteEn<lsu>	85% of PCLK

Table 35–149. RAM Interface Output Signal Timing (continued)

Output Signal Name	Td_max
DRam<did>WrData<lsu>	85% of PCLK
DRam<did>CheckWrData<lsu>	85% of PCLK
DRam<did>Lock<lsu>	85% of PCLK
IRam<iid>Addr	85% of PCLK
IRam<iid>En	80% of PCLK
IRam<iid>Wr	85% of PCLK
IRam<iid>WordEn	85% of PCLK
IRam<iid>WrData	85% of PCLK
IRam<iid>CheckWrData	85% of PCLK
IRam<iid>LoadStore	70% of PCLK

Table 35–150. RAM Interface Input Signal Timing

Input Signal Name	Set up (5-stage pipeline)	Set up (7-stage pipeline)
DRam<did>Data<lsu>	40% of PCLK	85% of PCLK
DRam<did>CheckData<lsu>	40% of PCLK	85% of PCLK
DRam<did>Busy<lsu>	85% of PCLK	85% of PCLK
IRam<iid>Data	40% of PCLK	85% of PCLK
IRam<iid>CheckData	40% of PCLK	85% of PCLK
IRam<iid>Busy	85% of PCLK	85% of PCLK

If the data RAM interface is opened to be used by an external customer-designed connection box, the Xtensa processor has different data RAM interface that splits read and write ports for design flexibility. Refer to Section 18.10 “Exposed Processor Interface for a Customer-Designed C-Box” for the details of this interface. For split read/write ports, the signal timing is summarized in Table 35–151 and Table 35–152. In the following tables, <did> refers to the DRam ID, which can take the value 0 or 1.

<lsu> refers to the load/store unit number (0/1) or indicates inbound PIF requests (DMA).

Table 35–151. RAM Output Signal Timing for the Split Read/Write Interface

Output Signal Name	Td_max
DRam<did>Addr<lsu>	85% of PCLK
DRam<did>En<lsu>	85% of PCLK
DRam<did>ByteEn<lsu>	85% of PCLK
DRam<did>WrAddr<lsu>	50% of PCLK
DRam<did>Wr<lsu>	50% of PCLK
DRam<did>WrByteEn<lsu>	50% of PCLK
DRam<did>WrData<lsu>	50% of PCLK
DRam<did>CheckWrData<lsu>	65% of PCLK
DRam<did>Lock<lsu>	85% of PCLK
DmaHighPriority	85% of PCLK

Table 35–152. RAM Input Signal Timing for the Split Read/Write Interface

Input Signal Name	Set up (5-stage pipeline)	Set up (7-stage pipeline)
DRam<did>Data<lsu>	40% of PCLK	85% of PCLK
DRam<did>CheckData<lsu>	40% of PCLK	85% of PCLK
DRam<did>Busy<lsu>	85% of PCLK	85% of PCLK
DRam<did>WrBusy<lsu>	85% of PCLK	85% of PCLK

Table 35–153. RAM Interface Input Signal Timing

Input Signal Name	Set up
DRam<did>Data0	40% of PCLK
DRam<did>CheckData0	40% of PCLK
DRam<did>Busy0	85% of PCLK
IRam<iid>Data	40% of PCLK
IRam<iid>CheckData	40% of PCLK
IRam<iid>Busy	80% of PCLK

35.8 ROM Interface AC Timing

Table 35–154. ROM Interface Output Signal Timing

Output Signal Name	Td_max
DRom0Addr<lsu>	85% of PCLK
DRom0En<lsu>	85% of PCLK
DRom0ByteEn<lsu>	85% of PCLK
IRom0Addr	85% of PCLK
IRom0En	85% of PCLK
IRom0WordEn	85% of PCLK
IRom0Load	70% of PCLK

35.9 XLMI Interface AC Timing

Table 35–155. ROM Interface Input Signal Timing

Input Signal Name	Set up (5-stage pipeline)	Set up (7-stage pipeline)
DRom0Data<lsu>	40% of PCLK	85% of PCLK
DRom0Busy<lsu>	85% of PCLK	85% of PCLK
IRom0Data	40% of PCLK	85% of PCLK
IRom0Busy	85% of PCLK	85% of PCLK

Table 35–156. XLMI Output Signal Timing

Output Signal Name	Td_max
DPort0Addr<lsu>	85% of PCLK
DPort0ByteEn<lsu>	85% of PCLK
DPort0En<lsu>	85% of PCLK
DPort0Wr<lsu>	85% of PCLK
DPort0WrData<lsu>	85% of PCLK
DPort0Load<lsu>	70% of PCLK
DPort0LoadRetired<lsu>	50% of PCLK
DPort0RetireFlush<lsu>	50% of PCLK

Table 35–157. XLMI Input Signal Timing

Input Signal Name	Set up (5-stage pipeline)	Set up (7-stage pipeline)
DPort0Data<lsu>	40% of PCLK	85% of PCLK
DPort0Busy<lsu>	85% of PCLK	85% of PCLK

35.10 Prefetch Signal AC Timing

Note that the prefetch memory is always a single cycle flow through memory (the 5-stage memory), where the outputs are taken directly from the memory and not registered externally, as for the pipelined memory (7-stage memory). See Section 9.1 “Core Microarchitecture Options” for details about pipelines. Table 35–158 shows the default AC timing for extra prefetch memory inputs and outputs that are present with the Prefetch option. For more precise timing, use the timing numbers from your actual memories.

Table 35–158. Prefetch Memory Default AC Timing

Signal	Input Setup (Tsu)	Output Delay (Td_max)
PrefetchRamEn		85% of PCLK
PrefetchRamPIFWEn		85% of PCLK
PrefetchRamWr		85% of PCLK
PrefetchRamAddr		85% of PCLK
PrefetchRamWrData		85% of PCLK
PrefetchRamData	40% of PCLK	

35.11 Combinational-Logic-Only Signal Paths on Interface Ports

The Xtensa processor has combinational-logic-only signal paths from the following interface-port inputs:

- I<*type*><*id*>Busy, where
  - <*type*> is Ram or Rom
  - <*id*> is 0/1
- D<*type*><*id*>Busy<*lsu*>, where:
  - <*type*> is Ram, Rom, or Port
  - <*id*> is 0/1
  - <*lsu*> designates the load/store unit (0/1/DMA) or bank ID

To all of the following outputs:

- +, where:
  - <*type*> is Tag, Cache, Ram, or Rom
  - <*way*> is A/B/C/D
  - <*id*> is 0/1
  - <output> is En, Addr, WordEn, Wr, or WrData
- D<*type*><*way/id*><*output*><*lsu*>, where:
  - <*type*> is Tag, Cache, Ram, Rom, or Port
  - <*way*> is A/B/C/D
  - <*id*> is 0/1
  - <*lsu*> designates the load/store unit (0/1/DMA) or bank ID
  - <*output*> is En, Addr, ByteEn, Wr, WrData, Load, LoadRetired, or RetireFlush
- TIE_<*name*>_PopReq, where:
  - <*name*> is the name of an input queue.

Paths from all inputs to all outputs exist individually and separately (for example, from IRam0Busy to DCacheBWrData0).

Note: The Xtensa processor configurations without Busy's have fully synchronous interfaces.

In asynchronous reset configurations, there are additional combinational-only logic paths as follows:

- BReset to PReset
- TMode to PReset

35.12 TIE Queue and Port Signal Timing

Note: Both the pre-configured GPIO32 Ports and QIF32 Queues, and the user-defined TIE Ports and Queues have the same AC Timing.

Table 35–159. TIE Input Queue Timing (per input queue instantiation)

Signal Name	Signal Type (Input or Output)	Setup (Tsu)	Td_max
TIE_<name>	Input	30% of PCLK	
TIE_<name>_PopReq	Output		70% of PCLK
TIE_<name>_Empty	Input	50% of PCLK	

Notes:

There is one set of these three interfaces per input queue instantiation.

TIE code: queue <name> <iqwifth> in

Table 35–160. TIE Output Queue Timing (per output queue instantiation)

Signal Name	Signal Type (Input or Output)	Setup (Tsetup)	Td_max
TIE_<name>	Output		30% of PCLK
TIE_<name>_PushReq	Output		70% of PCLK
TIE_<name>_Full	Input	50% of PCLK	

Notes:

There is one set of these three interfaces per output queue instantiation.

TIE code: queue <name> <oqwifth> out

Table 35–161. TIE Lookup Timing (per TIE Lookup instantiation)

Signal Name	Signal Type (Input or Output)	Setup (Tsetup)	Td_max
TIE_<name>_Out_Req	Output		30% of PCLK
TIE_<name>_Out	Output		70% of PCLK
TIE_<name>_In	Input	50% of PCLK	
TIE_<name>_Rdy	Input	50% of PCLK	

Notes:

There is one set of these interfaces per lookup instantiation.

TIE code: lookup <name> {<owidth>, <ostage>} {<iwidth>, <istage>} [rdy]

Table 35–162. TIE Export State Timing (per exported state)

Output Signal Name	Td_max
TIE_<name>	30% of PCLK
Note:	
There is one interface per state that is exported.	
TIE code: state <name> <statewidth> export	

Table 35–163. TIE Import Wire Timing (per wire)

Input Signal Name	Setup (Tsetup)
TIE_<name>	30% of PCLK
Note:	
There is one interface per imported wire.	
TIE code: import_wire <name> <width>	

A. Xtensa Pipeline and Memory Performance

The following sections detail the Xtensa processor's pipeline and memory performance. Numerous factors affect performance, including:

- Data and resource dependencies
- Branch penalties
- Instruction- and data-cache miss penalties

An understanding of these factors is crucial to achieving maximum performance from an Xtensa processor.

Figure A–193 and Figure A–194 show the operational behavior of a 5-stage and 7-stage pipeline, respectively. The 5-stage pipeline consists of the I stage (instruction fetch), the R stage (register-read), the E stage (execute), the M stage (memory access), and the W stage (register-write). The 7-stage pipeline has an extra cycle of latency for both the instruction-memory fetch and for the data-memory fetch, which adds an H stage just before the I stage and the L stage just before the M stage. In some cases, it is convenient to add a 6th (or 8th) stage called the P stage, corresponding to the program counter (PC) address generation, but this stage almost always overlaps other stages and so it is not generally illustrated.

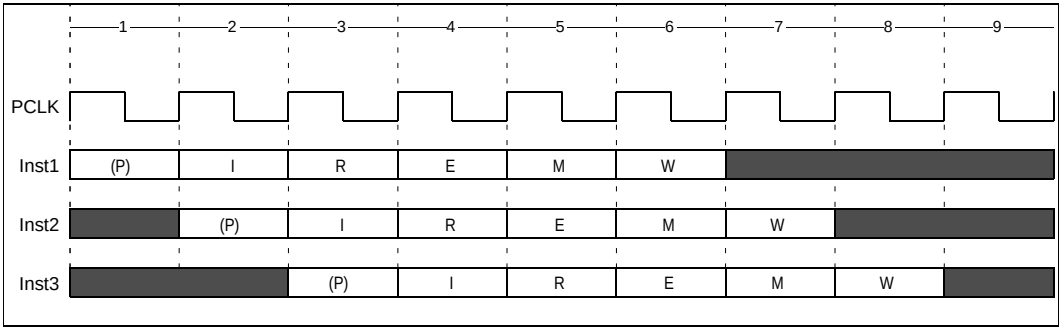


Figure A–193. Operation of the 5-Stage Xtensa LX7 Pipeline

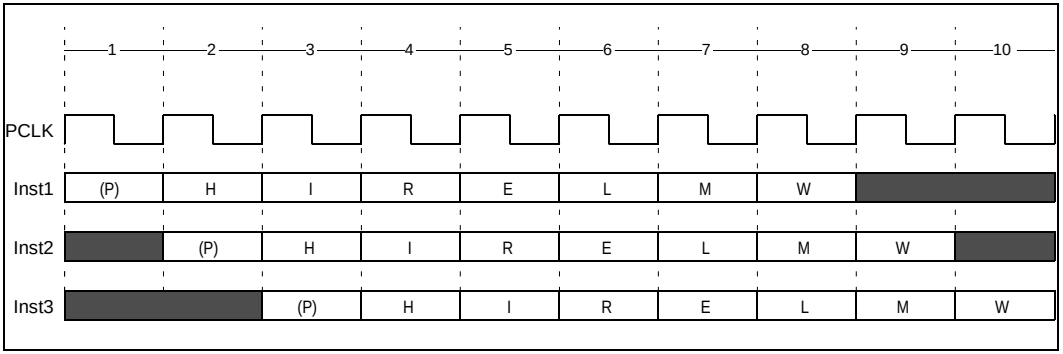


Figure A–194. Operation of the 7-Stage Xtensa LX7 Pipeline

Figure A–195 shows the relationship of the scalar pipeline to the vector DSP coprocessor pipeline's offered as standard options of the Xtensa LX processor. The DSP pipeline generally begins coincident with the M-stage of the pipeline. The M-stage is selected to minimize the latency between a data load and DSP operation. Most fixed-point DSP's have a single-cycle latency for basic operations, and longer latency for multiply-accumulate and some complex operations. FLIX allows bundling of scalar and vector operations into a single instruction.

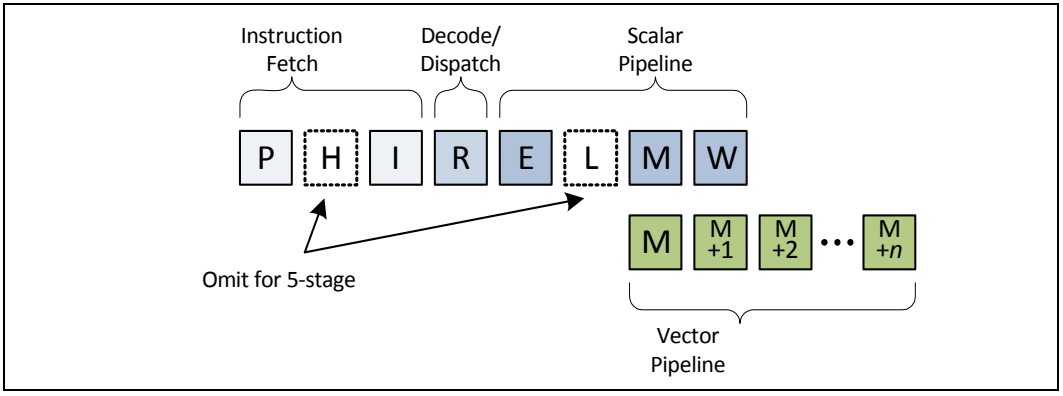


Figure A–195. Relationship of Scalar Pipeline to Vector DSP Pipeline

A.1 Pipeline Data and Resource Dependency Bubbles

An instruction is held in the R stage whenever it has a pipeline data or resource dependency with an instruction in the later pipeline stages. These cases are detailed in the following subsections.

A.1.1 Instruction Latency and Throughput

It is important to have an accurate model of processor performance for both code generation and simulation. However, the interactions of multiple instructions in a processor pipeline can be complex. It is common to simplify and describe pipeline and cache performance separately even though they may interact, because the information is used in different stages of compilation or coding. We adopt this approach, and then separately describe some of the interactions.

It is also common to describe the pipelining of instructions with numbers for *latency* (the time an instruction takes to produce its result after it receives its inputs) and *throughput* (the time an instruction delays other instructions independent of operand dependencies), but this simplified model cannot accommodate some situations. Therefore, we adopt a slightly more complicated, and more accurate, model. This model predicts when one instruction *issues* relative to other instructions. An instruction issues when all of its data inputs are available and all the necessary hardware functional units are available for it to proceed. The point of instruction issue is when computation of the instruction's results begins.

Instruction Operand Dependency

Instead of using a per-instruction latency number, instructions are modeled as taking their operands in various pipeline stages and producing results in other pipeline stages. When instruction Inst1 writes (or defines) operand X (either an explicit operand or implicit state register) and instruction Inst2 reads (or uses) operand X, then instruction Inst2 depends on Inst1.[1]

If instruction Inst1 defines operand X in stage SA (at the end of the stage), and instruction Inst2 uses operand X in stage SB (at the beginning of the stage), then instruction Inst2 can issue no earlier than D = max(SA − SB + 1, 0) cycles after Inst1 issues. This is illustrated in Figure A–196, where SA is 3 and SB is 1, hence the number of cycles needed is 3.

If the processor reaches instruction Inst2 earlier than D cycles after instruction Inst1, it generally delays Inst2's issue into the pipeline until D cycles have elapsed. When the processor delays an instruction because of a pipeline interaction, the delay mechanism is called an *interlock*. For a few special dependencies (primarily those involving the special registers controlling exceptions, interrupts, and memory management) the processor does not interlock. These situations are called *hazards*. For correct operation, code generation by the Xtensa C/C++ compiler must insert `xSYNC` instructions to avoid hazards by delaying the dependent instruction. The `xSYNC` instructions accomplishes this delay in an implementation-independent manner.

1. This situation is called a "read-after-write" dependency. Other possible operand dependencies familiar to coders are "write-after-write" and "write-after-read," but these other dependencies have no pipeline performance implications in the Xtensa LX7 6 processor.

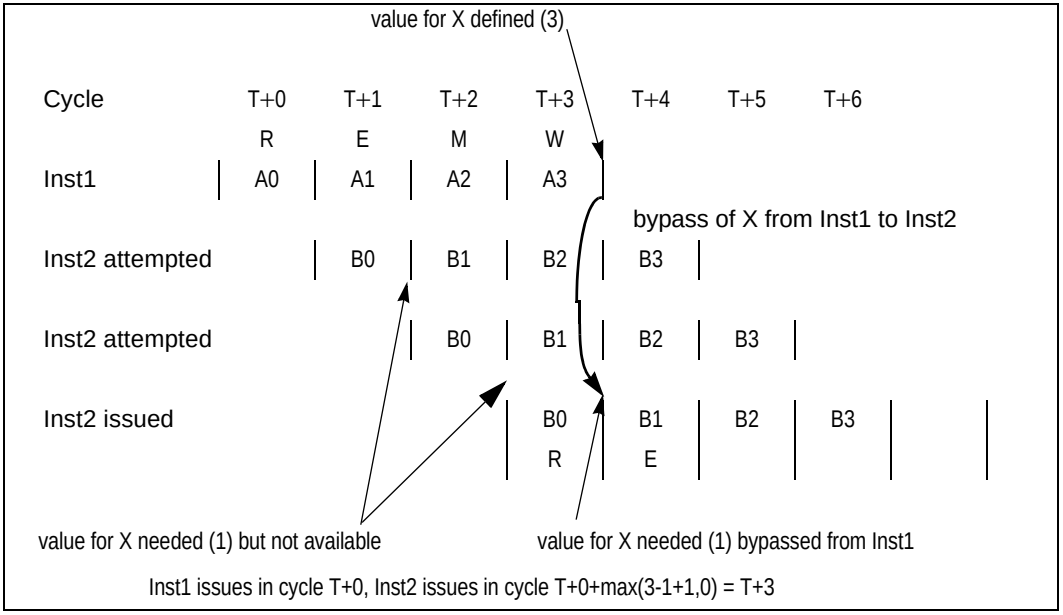


Figure A–196. Instruction Operand Dependency Interlock

Operand Bypassing

When we describe an instruction as making one of its values available at the end of a stage, this description refers to when the computation is complete and not necessarily the time that the processor state is written. It is usual to delay the writing of the state until at least the point at which the instruction is *committed* (that is, cannot be aborted by its own or an earlier instruction's exception).

In some implementations, the processor delays the writing of the state still further to satisfy resource constraints. However, the delay in writing the processor state is usually invisible; Xtensa processors detect the use of an operand that has been produced by one instruction by another instruction even though the processor state has not yet been written and they directly forward the required value from one pipeline stage to the other. This operation is called a *bypass*.

Table A–164 assigns numbers to various pipeline stages for both the 5- and 7-stage Xtensa pipelines.

This table describes each instruction's pipeline characteristics using those assigned stage numbers. The table has one row per instruction, and columns that give pipeline stage numbers for the input and output of both explicit operands and implicit operands. The stage numbers begin at 0 for the R stage and increment for each successive pipeline stage.

Table A–164. Stage Numbering

Pipeline	0	1	2	3	4
5-stage	R	E	M	W	
7-stage	R	E	L	M	W

Iterative Implementations

Some options are or can be implemented iteratively to reduce the amount of hardware required to implement them. For example, the Integer Divide option is always implemented iteratively. The 32-bit Integer Multiply Option can be optionally implemented iteratively. Iterative functions use a small arithmetic block that is cycled multiple times over consecutive cycles to perform the operation. The Integer Divide option adds a logic block to the processor's execution unit that generates 3-bits of quotient per cycle and thus requires as many as 11 passes or iterations to perform a 32-bit divide. During its execution stage, the iterative-divide instruction freezes the processor's pipeline. (See Appendix A.9 “GlobalStall Causes” for more details on GlobalStall which the iterative-divide logic uses to freeze the pipeline).

The integer multiplier options including the 32-bit Integer Multiplier option’s MULL instruction and the 16-bit Integer Multiplier option’s MUL16U and MUL16S instructions can be optionally implemented either iteratively or as fully-pipelined instructions.

Iterative implementations extend instruction latency but do not affect operand scheduling or bypassing as discussed below.

Instruction Pipeline Dependency Tables

As described earlier, any pair of instructions A and B with a read-after-write dependency on X must be separated in the pipeline by at least max(AXdefstage − BXusestage + 1, 0) cycles. *Define* stage numbers represent the stage number in which the result is complete and will be available for use as an input operand (via bypass) on the following cycle. This is likely different from the stage number in which the processor state is written, which typically happens one or more stages later. For example, an instruction with *Use* specification “`as` 1” and *Define* specification “`ar` 2” requires its `as` (source) input at the beginning of stage 1, and makes its `ar` (destination) output available at the end of stage 2. The latency is 2 cycles (2 − 1 + 1 = 2).

Instructions defined in TIE may optionally specify equivalent *use* and *define* stage numbers for instructions via TIE’s `schedule` declaration. A default stage assignment is made if no `schedule` declaration is given. Default stage assignments for TIE instructions are the beginning of stage 1 for inputs to be read (*used*) and the end of stage 1 (that is, similar to the ADD instruction) for outputs to be available (*defined*).

Memory interfaces are an exception to this default rule, and are defined to the stage corresponding to the M stage (stage 2 for a 5-stage pipeline and stage 3 for a 7-stage pipeline).

The tables do not attempt to represent the effects of instruction or data-operand accesses that cause cache misses.

Note: When a *use* or *define* schedule differs for a 5-stage and 7-stage pipeline, a '/' will be used to distinguish them. For example, 2/3 would indicate stage 2 for a 5-stage pipeline and stage 3 for a 7-stage pipeline. These numbers represent the M stage in both cases.

Table A–165. Xtensa Processor Family Instruction Pipelining

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
ABS	at 1	ar 1	—	—
ABS.D	frs 2/3	frr 2/3	—	—
ABS.S	frs 2/3	frr 2/3	—	—
ADD	as 1, at 1	ar 1	—	—
ADD.N	as 1, at 1	ar 1	—	—
ADD.D	frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5, FSR 5/6	FSR 5/6
ADD.S	frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5, FSR 5/6	FSR 5/6
ADDEXP.D	frs 2/3, frt 2/3	frr 2/3	—	—
ADDEXP.S	frs 2/3, frt 2/3	frr 2/3	—	—
ADDEXPM.D	frs 2/3, frt 2/3	frr 2/3	—	—
ADDEXPM.S	frs 2/3, frt 2/3	frr 2/3	—	—
ADDI	as 1	at 1	—	—
ADDI.N	as 1	ar 1	—	—
ADDMI	as 1	at 1	—	—
ADDX2	as 1, at 1	ar 1	—	—
ADDX4	as 1, at 1	ar 1	—	—
ADDX8	as 1, at 1	ar 1	—	—
AND	as 1, at 1	ar 1	—	—
ALL4	bs 1	bt 1	—	—
ALL8	bs 1	bt 1	—	—
AND	as 1, at 1	ar 1	—	—

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
ANDB	bs 1, bt 1	br 1	—	—
ANDBC	bs 1, bt 1	br 1	—	—
ANY4	bs 1	bt 1	—	—
ANY8	bs 1	bt 1	—	—
BALL	as 1, at 1	—	—	—
BANY	as 1, at 1	—	—	—
BBC	as 1, at 1	—	—	—
BBCI	as 1	—	—	—
BBS	as 1, at 1	—	—	—
BBSI	as 1	—	—	—
BEQ	as 1, at 1	—	—	—
BEQI	as 1	—	—	—
BEQT	as 1, at 1	—	—	—
BEQZ	as 1	—	—	—
BEQZ.N	as 1	—	—	—
BEQZT	at 1	—	—	—
BF	bs 1	—	—	—
BGE	as 1, at 1	—	—	—
BGEI	as 1	—	—	—
BGEU	as 1, at 1	—	—	—
BGEUI	as 1	—	—	—
BGEZ	as 1	—	—	—
BLT	as 1, at 1	—	—	—
BLTI	as 1	—	—	—
BLTU	as 1, at 1	—	—	—
BLTUI	as 1	—	—	—
BLTZ	as 1	—	—	—
BNALL	as 1, at 1	—	—	—
BNE	as 1, at 1	—	—	—
BNEI	as 1	—	—	—
BNET	as 1, at 1	—	—	—
BNEZ	as 1	—	—	—
BNEZ.N	as 1	—	—	—

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
BNEZT	at 1	—	—	—
BNONE	as 1, at 1	—	—	—
BREAK	—	—	—	—
BREAK.N	—	—	—	—
BT	bs 1	—	—	—
CALL0	—	—	—	AR[0] 1
CALL4	—	—	—	AR[4] 1
CALL8	—	—	—	AR[8] 1
CALL12	—	—	—	AR[12] 1
CALLX0	as 1	—	—	AR[0] 1
CALLX4	as 1	—	—	AR[4] 1
CALLX8	as 1	—	—	AR[8] 1
CALLX12	as 1	—	—	AR[12] 1
CEIL.D	frs 2/3	arr 3/4	FSR 5/6	FSR 5/6
CEIL.S	frs 2/3	arr 3/4	FSR 5/6	FSR 5/6
CLAMPS	as 1	ar 1	—	—
CLREX	—	—	—	—
CONST.D	—	frr 2/3	—	—
CONST.S	—	frr 2/3	—	—
CRC8	as 1, at 1	ar 1	—	—
DEPBITS	ar 1, as 1	ar 1		
DHI	as 1	—	—	—
DHI.B	as 1, at 1	—	—	—
DHU	as 1			
DHWB	as 1	—	—	—
DHWB.B	as 1, at 1	—	—	—
DHWBI	as 1	—	—	—
DHWBI.B	as 1, at 1	—	—	—
DII	as 1	—	—	—
DIU	as 1			
DIVN.D	frr 2/3, frs 2/3, frt 2/3	frr 5/6	FSR 5/6	FCR.RoundMode 4/5, FSR 5/6

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
DIVN.S	frr 2/3, frs 2/3, frt 2/3	frr 5/6	FSR 5/6	FCR.RoundMode 4/5, FSR 5/6
DIV0.D	frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
DIV0.S	frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
DIWB	as 1	—	—	—
DIWBI	as 1	—	—	—
DIWBUI.P	as 1	as 1	—	—
DPFL	as 1	—	—	—
DPFM.B	as 1, at 1	—	—	—
DPFM.BF	as 1, at 1	—	—	—
DPFR	as 1			
DPFR.B	as 1, at 1	—	—	—
DPFR.BF	as 1, at 1	—	—	—
DPFRO	as 1			
DPFW	as 1			
DPFW.B	as 1, at 1	—	—	—
DPFW.BF	as 1, at 1	—	—	—
DPFWO	as 1			
DSYNC	—	—	—	—
ENTRY	as 1	as 1	—	—
ESYNC	—	—	—	—
EXCW	—	—	—	—
EXTUI	at 1	ar 1	—	—
FLOAT.D	ars 1	frr 3/4	FCR.RoundMode 2/3, FSR 5/6	FSR 5/6
FLOAT.S	ars 1	frr 3/4	FCR.RoundMode 2/3, FSR 5/6	FSR 5/6
FLOOR.D	frs 2/3	arr 3/4	FSR 5/6	FSR 5/6
FLOOR.S	frs 2/3	arr 3/4	FSR 5/6	FSR 5/6
GETEX	at 1	at 1		
IDTLB	as 1	—	—	—
IHI	as 1	—	—	—
IHU	as 1	—	—	—
III	as 1	—	—	—

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
IITLB	as 1	—	—	—
IIU	as 1	—	—	—
INVAL	not implemented – illegal instruction			
IPF	as 1			
IPFL	as 1	—	—	—
ISYNC	—	—	—	—
J	—	—	—	—
JX	as 1	—	—	—
L8UI	as 1	at 2	—	—
L16SI	as 1	at 2	—	—
L16UI	as 1	at 2	—	—
L32AI	as 1	at 2		
L32E	as 1	at 2/3	—	—
L32EX	as 1	at 2/3	—	—
L32I	as 1	at 2	—	—
L32I.N	as 1	at 2	—	—
L32R	—	at 2	—	—
L32SI	not implemented – illegal instruction			
LDCT	as 1	at 2/3	—	—
LDCW	as 1	at 2/3	—	—
LDDEC	as 1	mw 2, as 1	—	—
LDDR32.P	as 1	as 1	PSRING 2/3, PSEXCM 2/3, InOCDMode 1	—
LDI	ars 1	frt 2/3	—	—
LDINC	as 1	mw 2, as 1	—	—
LDIP	ars 1	frt 2/3, ars 1	—	—
LDX	ars 1, art 1	frr 2/3	—	—
LDXP	ars 1, art 1	frr 2/3, ars 1	—	—
LICT	as 1	at 2/3	—	—
LICW	as 1	at 2/3	—	—
LICT	as 1	at 2	—	—
LICW	as 1	at 2	—	—

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
LOOP	as 1	—	—	—
LOOPGTZ	as 1	—	—	—
LOOPNEZ	as 1	—	—	—
LSI	ars 1	frt 2/3	—	—
LSIP	ars 1	frt 2/3, ars 1	—	—
LSX	ars 1, art 1	frr 2/3	—	—
LSXP	ars 1, art 1	frr 2/3, ars 1	—	—
MADD.D	frr 2/3, frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5, FSR	FSR 5/6
MADD.S	frr 2/3, frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5, FSR	FSR 5/6
MADDN.D	frr 2/3, frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5, FSR	FSR 5/6
MADDN.S	frr 2/3, frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5, FSR	FSR 5/6
MAX	as 1, at 1	ar 1	—	—
MAXU	as 1, at 1	ar 1	—	—
MEMW	—	—	—	—
MIN	as 1, at 1	ar 1	—	—
MINU	as 1, at 1	ar 1	—	—
MKDADJ.D	frr 2/3, frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
MKDADJ.S	frr 2/3, frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
MKSADJ.D	frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
MKSADJ.S	frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
MOV.N	as 1	at 1	—	—
MOV.D	frs 2/3	frr 2/3	—	—
MOV.S	frs 2/3	frr 2/3	—	—
MOVEQZ	as 1, at 1	ar 1	—	—
MOVEQZ.D	frs 2/3, art 1	frr 2/3	—	—
MOVEQZ.S	frs 2/3, art 1	frr 2/3	—	—
MOVF	as 1, bt 1	ar 1	—	—
MOVF.D	frs 2/3, bt 1	frr 2/3	—	—
MOVF.S	frs 2/3, bt 1	frr 2/3	—	—
MOVGEZ	as 1, at 1	ar 1	—	—

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
MOVGEZ.D	frs 2/3, art 1	frr 2/3	—	—
MOVGEZ.S	frs 2/3, art 1	frr 2/3	—	—
MOVI	—	at 1	—	—
MOVI.N	—	as 1	—	—
MOVLTZ	as 1, at 1	ar 1	—	—
MOVLTZ.D	frs 2/3, art 1	frr 2/3	—	—
MOVLTZ.S	frs 2/3, art 1	frr 2/3	—	—
MOVNEZ	as 1, at 1	ar 1	—	—
MOVNEZ.D	frs 2/3, art 1	frr 2/3	—	—
MOVNEZ.S	frs 2/3, art 1	frr 2/3	—	—
MOVSP	as 1	at 1	—	—
MUL16S	as 1, at 1	ar 2	—	—
MUL16U	as 1, at 1	ar 2	—	—
MOVT	as 2/3, bt 1	ar 2/3	—	—
MOVT.D	frs 2/3, bt 1	frr 2/3	—	—
MOVT.S	frs 2/3, bt 1	frr 2/3	—	—
MSUB.D	frr 2/3, frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5, FSR 5/6	FSR 5/6
MSUB.S	frr 2/3, frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5, FSR 5/6	FSR 5/6
MUL.D	frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5. FSR 5/6	FSR 5/6
MUL.S	frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5. FSR 5/6	FSR 5/6
MUL.AA.HH	as 1, at 1	—	—	ACCLO 2, ACCHI 2
MUL.AA.HL	as 1, at 1	—	—	ACCLO 2, ACCHI 2
MUL.AA.LH	as 1, at 1	—	—	ACCLO 2, ACCHI 2
MUL.AA.LL	as 1, at 1	—	—	ACCLO 2, ACCHI 2
MUL.AD.HH	as 1, my 1	—	—	ACCLO 2, ACCHI 2

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
MUL.AD.HL	as 1, my 1	—	—	ACCLO 2, ACCHI 2
MUL.AD.LH	as 1, my 1	—	—	ACCLO 2, ACCHI 2
MUL.AD.LL	as 1, my 1	—	—	ACCLO 2, ACCHI 2
MUL.DA.HH	mx 1, at 1	—	—	ACCLO 2, ACCHI 2
MUL.DA.HL	mx 1, at 1	—	—	ACCLO 2, ACCHI 2
MUL.DA.LH	mx 1, at 1	—	—	ACCLO 2, ACCHI 2
MUL.DA.LL	mx 1, at 1	—	—	ACCLO 2, ACCHI 2
MUL.DD.HH	mx 1, my 1	—	—	ACCLO 2, ACCHI 2
MUL.DD.HL	mx 1, my 1	—	—	ACCLO 2, ACCHI 2
MUL.DD.LH	mx 1, my 1	—	—	ACCLO 2, ACCHI 2
MUL.DD.LL	mx 1, my 1	—	—	ACCLO 2, ACCHI 2
MUL16S	as 1, at 1	ar 1/2 [1]	—	—
MUL16U	as 1, at 1	ar 1/2 [1]	—	—
MULA.AA.HH	as 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.AA.HL	as 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.AA.LH	as 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.AA.LL	as 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.AD.HH	as 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.AD.HL	as 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.AD.LH	as 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
MULA.AD.LL	as 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.HH	mx 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.HH.LDDEC	as 1, mx 1, at 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.HH.LDINC	as 1, mx 1, at 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.HH.LDDEC	as 1, mx 1, at 1	mw 2, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.HH.LDINC	as 1, mx 1, at 1	mw 2, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.HL	mx 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.HL.LDDEC	as 1, mx 1, at 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.HL.LDINC	as 1, mx 1, at 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.HL.LDDEC	as 1, mx 1, at 1	mw 2, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.HL.LDINC	as 1, mx 1, at 1	mw 2, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.LH	mx 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.LH.LDDEC	as 1, mx 1, at 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.LH.LDINC	as 1, mx 1, at 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.LH.LDDEC	as 1, mx 1, at 1	mw 2, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.LH.LDINC	as 1, mx 1, at 1	mw 2, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.LL	mx 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.LL.LDDEC	as 1, mx 1, at 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DA.LL.LDINC	as 1, mx 1, at 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
MULA.DD.HH	mx 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DD.HH.LDDEC	as 1, mx 1, my 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DD.HH.LDINC	as 1, mx 1, my 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DD.HL	mx 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DD.HL.LDDEC	as 1, mx 1, my 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DD.HL.LDINC	as 1, mx 1, my 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DD.LH	mx 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DD.LH.LDDEC	as 1, mx 1, my 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DD.LH.LDINC	as 1, mx 1, my 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DD.LL	mx 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DD.LL.LDDEC	as 1, mx 1, my 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULA.DD.LL.LDINC	as 1, mx 1, my 1	mw 2/3, as 1	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.AA.HH	as 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.AA.HL	as 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.AA.LH	as 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.AA.LL	as 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.AD.HH	as 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.AD.HL	as 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.AD.LH	as 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
MULS.AD.LL	as 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.DA.HH	mx 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.DA.HL	mx 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.DA.LH	mx 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.DA.LL	mx 1, at 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.DD.HH	mx 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.DD.HL	mx 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.DD.LH	mx 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULS.DD.LL	mx 1, my 1	—	ACCLO 2, ACCHI 2	ACCLO 2, ACCHI 2
MULL	as 1, at 1	ar 1/2 [1]	—	—
MULSH	as 1, at 1	ar 2	—	—
MULUH	as 1, at 1	ar 2	—	—
NEG	at 1	ar 1	—	—
NEG.D	frs 2/3	frr 2/3	—	—
NEG.S	frs 2/3	frr 2/3	—	—
NEXP01.D	frs 2/3	frr 2/3	—	—
NEXP01.S	frs 2/3	frr 2/3	—	—
NOP	—	—	—	—
NOP.N	—	—	—	—
NSA	as 1	at 1	—	—
NSAU	as 1	at 1	—	—
OEQ.D	frs 2/3, frt 2/3	br 2/3	—	FSR 5/6
OEQ.S	frs 2/3, frt 2/3	br 2/3	—	FSR 5/6
OLE.D	frs 2/3, frt 2/3	br 2/3	—	FSR 5/6
OLE.S	frs 2/3, frt 2/3	br 2/3	—	FSR 5/6
OLT.D	frs 2/3, frt 2/3	br 2/3	—	FSR 5/6

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
OLT.S	frs 2/3, frt 2/3	br 2/3	—	FSR 5/6
OR	as 1, at 1	ar 1	—	—
ORB	bs 1, bt 1	br 1	—	—
ORBC	bs 1, bt 1	br 1	—	—
PDTLB	as 1	at 2/3	—	—
PFEND.A	—	—	—	—
PFEND.O	—	—	—	—
PFNXT.F	—	—	—	—
PFWAIT.A	—	—	—	—
PFWAIT.R	—	—	—	—
PITLB	as 1	at 2/3	—	—
PPTLB	as 1	at 2/3	—	—
POPC	as 1	at 1	—	—
QUOS	as 1, at 1	ar 1	—	—
QUOU	as 1, at 1	ar 1	—	—
RDTLB0	as 1	at 2/3	—	—
RDTLB1	as 1	at 2/3	—	—
RECIP0.D	frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
RECIP0.S	frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
REMS	as 1, at 1	ar 1	—	—
REMU	as 1, at 1	ar 1	—	—
RER	as (varies by config)	at (varies by config)	—	—
RET	—	—	AR[0] 1	—
RET.N	—	—	AR[0] 1	—
RETW	—	—	AR[0] 1	—
RETW.N	—	—	AR[0] 1	—
RFDD	—	—	InOCDMode 1	InOCDMode 3/4
RFDO	—	—	InOCDMode 1	InOCDMode 3/4
RFE	—	—	—	—
RFI	—	—	—	—
RFUE	—	—	—	—
RFR	frs 1	arr 1	—	—

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
RFWO	—	—	—	—
RFWU	—	—	—	—
RITLB0	as 1	at 2/3	—	—
RITLB1	as 1	at 2/3	—	—
ROTW	—	—	—	—
ROUND.D	frs 2/3	arr 3/4	FSR 5/6	FSR 5/6
ROUND.S	frs 2/3	arr 3/4	FSR 5/6	FSR 5/6
RPTLB0	as 1	at 2/3	—	—
RPTLB1	as 1	at 2/3	—	—
RSIL	—	at 1	—	—
RSR	—	at 2	special register various	—
RSQRT0.D	frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
RSQRT0.S	frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
RUR	—	ar 1	user register 1	—
RSYNC	—	—	—	—
S8I	at 2/3, as 1	—	—	—
S16I	at 2/3, as 1	—	—	—
S32C1I	at 2/3, as 1	—	SCOMPARE1 3/4	—
S32E	at 2/3, as 1	—	—	—
S32EX	at 2/3, as 1	—	—	—
S32I	at 2/3, as 1	—	—	—
S32I.N	at 2/3, as 1	—	—	—
S32NB	at 2/3, as 1	—	—	—
S32RI	at 2/3, as 1	—	—	—
SALT	at 1, as 1	ar 1	—	—
SALTU	at 1, as 1	ar 1	—	—
SDCT	at 2/3, as 1	—	—	—
SDCW	at 2/3, as 1	—	—	—
SDDR32.P	as 1	as 1	PSRING 2/3, PSEXCM 2/3, InOCDMode 1	—
SDI	frt 2/3, ars 2	—	—	—
SDIP	frt 2/3, ars 2	ars 2	—	—

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
SDX	frr 2/3, ars 2, art 2	—	—	—
SDXP	frr 2/3, ars 2, art 2	ars 2	—	—
SEXT	as 1	ar 1	—	—
SICT	at 2/3, as 1	—	—	—
SICW	at 2/3, as 1	—	—	—
SLL	as 1	ar 1	SAR 1	—
SLLI	as 1	ar 1	—	—
SQRT0.D	frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
SQRT0.S	frs 2/3	frr 3/4	FSR 5/6	FSR 5/6
SRA	at 1	ar 1	SAR 1	—
SRAI	at 1	ar 1	—	—
SRC	as 1, at 1	ar 1	SAR 1	—
SRL	at 1	ar 1	SAR 1	—
SRLI	at 1	ar 1	—	—
SSA8B	as 1	—	—	SAR 1
SSA8L	as 1	—	—	SAR 1
SSAI	—	—	—	SAR 1
SSI	frt 2/3, ars 2	—	—	—
SSIP	frt 2/3, ars 2	ars 2	—	—
SSL	as 1	—	—	SAR 1
SSR	as 1	—	—	SAR 1
SSX	frr 2/3, ars 2, art 2	—	—	—
SSXP	frr 2/3, ars 2, art 2	ars 2	—	—
SUB	as 1, at 1	ar 1	—	—
SUB.D	frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5. FSR 5/6	FSR 5/6
SUB.S	frs 2/3, frt 2/3	frr 5/6	FCR.RoundMode 4/5. FSR 5/6	FSR 5/6
SUBX2	as 1, at 1	ar 1	—	—
SUBX4	as 1, at 1	ar 1	—	—
SUBX8	as 1, at 1	ar 1	—	—
SYSCALL	—	—	—	—
TRUNC.D	frs 2/3	arr 3/4	FSR 5/6	FSR 5/6

Instruction Mnemonic	Operand Stages		Implicit State Stages	
	Use	Define	Use	Define
TRUNC.S	frs 2/3	arr 3/4	FSR 5/6	FSR 5/6
UEQ.D	frs 2/3, frt 2/3	br 2/3	FSR.Invalid 5/6	FSR.Invalid 5/6
UEQ.S	frs 2/3, frt 2/3	br 2/3	FSR.Invalid 5/6	FSR.Invalid 5/6
UFLOAT.D	ars 1	frr 3/4	FCR.RoundMode 2/3. FSR 5/6	FSR 5/6
UFLOAT.S	ars 1	frr 3/4	FCR.RoundMode 2/3. FSR 5/6	FSR 5/6
ULE.D	frs 2/3, frt 2/3	br 2/3	FSR.Invalid 5/6	FSR.Invalid 5/6
ULE.S	frs 2/3, frt 2/3	br 2/3	FSR.Invalid 5/6	FSR.Invalid 5/6
ULT.D	frs 2/3, frt 2/3	br 2/3	FSR.Invalid 5/6	FSR.Invalid 5/6
ULT.S	frs 2/3, frt 2/3	br 2/3	FSR.Invalid 5/6	FSR.Invalid 5/6
UMUL.AA.HH	as 1, at 1	—	—	ACCLO 2, ACCHI 2
UMUL.AA.HL	as 1, at 1	—	—	ACCLO 2, ACCHI 2
UMUL.AA.LH	as 1, at 1	—	—	ACCLO 2, ACCHI 2
UMUL.AA.LL	as 1, at 1	—	—	ACCLO 2, ACCHI 2
UN.D	frs 2/3, frt 2/3	br 2/3	FSR.Invalid 5/6	FSR.Invalid 5/6
UN.S	frs 2/3, frt 2/3	br 2/3	FSR.Invalid 5/6	FSR.Invalid 5/6
UTRUNC.D	frs 2/3	arr 3/4	FSR 5/6	FSR 5/6
UTRUNC.S	frs 2/3	arr 3/4	FSR 5/6	FSR 5/6
WAITI	—	—	—	—
WDTLB	as 1, at 3/4	—	—	—
WER	as 3/4, at 3/4	—	—	—
WFR	ars 1	frr 1	—	—
WITLB	as 1, at 3/4	—	—	—
WPTLB	as 1, at 3/4	—	—	—
WSR	varies by special register	—	—	special register various
WUR	at 1	—	—	user register 1
XOR	as 1, at 1	ar 1	—	—
XORB	bs 1, bt 1	br 1	—	—

1. 1 for iterative implementations, 2 for fully-pipelined implementations

A.1.2 Load-Use Pipeline Bubbles

The most important data dependency case is probably the load-use case. A load-use dependency occurs when an instruction needs data from a load instruction, but the data is not available when it is needed because of the memory latency.

As detailed in the previous section, load instructions have their output defined in stage 2/3 (the M stage for both 5- and 7-stage pipelines). An example of such a dependency for a 5-stage pipeline is where a load instruction is immediately followed by an instruction that uses the data from the load.

Because the data is only available late in the load's M stage, and the instruction using the data needs the data in its E stage, a pipeline bubble must be inserted between the load and the use to accommodate the memory latency. In this case a bubble is inserted into the pipeline and the instruction waits one cycle in the R stage.

In a 5-stage pipeline, the memory address is generated in the E stage and the memory data is available one cycle later in the M stage. The result is a 1-cycle load-use penalty as shown in Figure A–197. Note that the bubble that occurs in the pipeline can be eliminated by inserting another instruction that does not use the result of the load between the load instruction and the instruction that uses the load data. The XCC compiler often makes such optimizations.

In a 7-stage pipeline, the memory address is generated in the E stage and the memory data is available two cycles later in the M stage. The result is a 2-cycle load-use penalty as shown in Figure A–198. Note that bubbles that occur in the pipeline can be eliminated by inserting two other instructions that do not use the result of the load between the load instruction and the instruction that uses the load data. The XCC compiler often makes such optimizations.

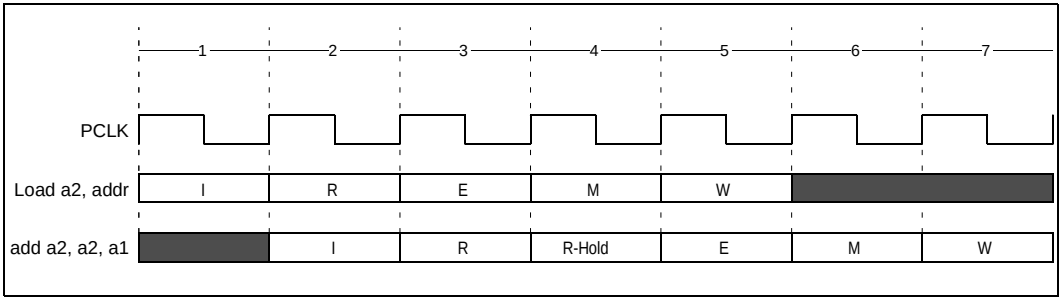


Figure A–197. 5-Stage Pipeline Load-Use Penalty

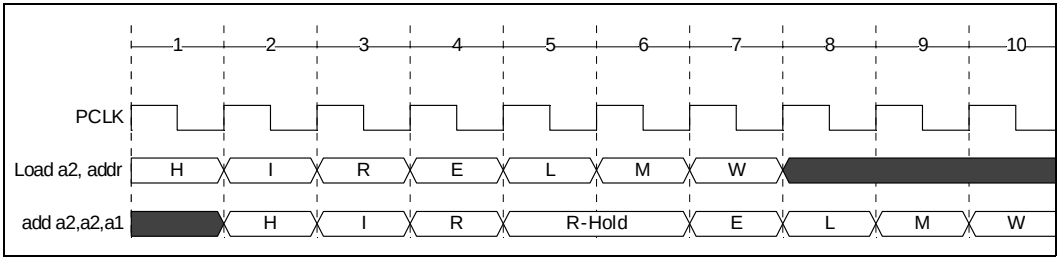


Figure A–198. 7-Stage Pipeline Load-Use Penalty

A.1.3 Cache Special Instruction Resource Hazard

As detailed in Section 9.7, a number of special cache instructions are added to the Xtensa processor if a cache option is selected. (The special instructions added depend on the cache sub-options selected such as locking or write-back cache.) To complete, these instructions tend to require atypical and often multipart cache accesses—in particular to the cache's tag array. The cache's data and tag arrays have only one read/write port, which leads to resource hazards between multiple cache-accessing instructions in the pipeline. These resource hazards are dealt with either by inserting bubbles in the pipeline or by replaying instructions.

In addition to resource hazards, there could be data dependencies between different special cache instructions in the pipeline. No bypassing or data forwarding has been implemented for these instructions, so additional bubbles are required because the processor must wait for one of these instructions to complete before it can let the next proceed. Figure A–199 demonstrates what happens to back-to-back DHI instructions.

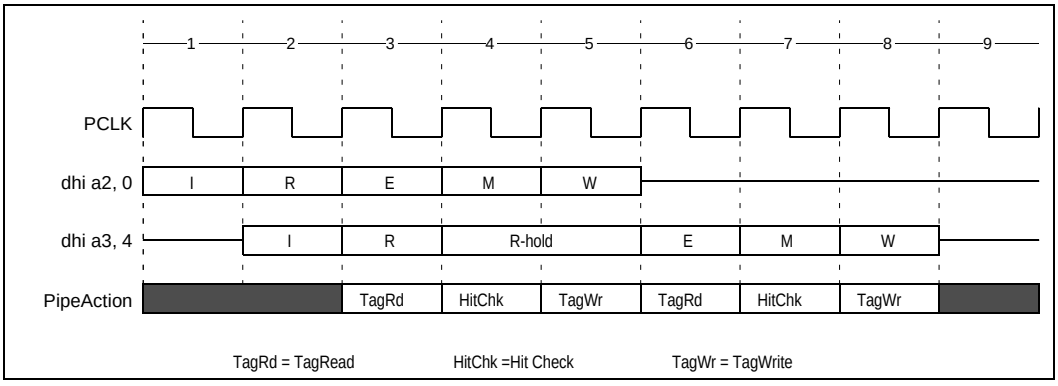


Figure A–199. Back-to-Back DHI Instructions

The second DHI is held in its R stage until the first instruction clears its last tag—accessing stage (its W stage). All special data-cache instruction hazards follow this rule. Any tag-writing special instruction (excluding dirty-bit-only writes performed by the DIWB

and DHWB instructions)—that is, the SDCT, DHWBI, DIWBI, DHI, DII, DPFL, DHU, and DIU instructions—that has not cleared its tag-writing stage W, causes any subsequent tag-reading instruction—that is, the load, store, LDCT, DHWB, DHWBI, DIWB, DIWBI, DHI, DII, DPFL, DHU, and DIU instructions—to be held in the R stage, so that the read of the younger instruction falls in the cycle immediately after the write of the older instruction.

The rules for resolving special instruction-cache instruction hazards are different and involve replays rather than bubbles. They are described later in Section A.4.

A.1.4 Other Data-Dependency Bubbles

Several instructions implement pipeline bubbles to resolve dependencies. These instructions include:

- The `MEMW` memory-barrier instruction is implemented by the processor holding it in the R stage until the pipeline, store buffer, and write buffer are flushed of all stores. In processor configurations with PIF write responses, `MEMW` will also be held until there are no outstanding write responses. In the fastest case, a `MEMW` will only take one cycle for itself.
- For SuperGather configurations, MEMW, EXCW, and EXTW wait for there to be no outstanding gather/scatter operations in the SuperGather engine.
- The exception-wait instruction, `EXCW`, is implemented similarly to a `MEMW` instruction. It allows stores to flush from the processor to allow any late exceptions from stores to be handled before committing the exception-wait instruction. In the fastest case, an `EXCW` will only take one cycle for itself.
- `EXTW` is implemented similarly to a `MEMW` instruction. In addition, this instruction is held in the R stage until all designer-defined TIE output queues are flushed and all writes to exported TIE states have committed (this is implemented by waiting for the pipeline to flush all valid instructions). In the fastest case, an `EXTW` will only take one cycle for itself.
- Release stalls are implemented for `S32RI` and `S32C1I`. These instructions are held in the processor's R pipeline stage under the same conditions as a `MEMW`. In the fastest case, there will be no additional bubbles.
- In addition to the release stalls, an `S32C1I` will also hold all subsequent loadstore and cache test instructions in the processor's R pipeline stage as long as the `S32C1I` is in the pipeline and the store buffers are not empty. This is done to ensure that no subsequent instructions access the target memory of the `S32C1I` until the `S32C1I` has completed.
- Pipeline stalls are implemented for `S32EX` and `L32EX` instructions. These instructions are held in the processor's R pipeline stage until the pipeline and store buffer are flushed of all stores. All exclusive instructions are held in the processor's R pipeline stage as long as there is a pending S32EX in the pipeline. In addition to the

stalls, pipeline E stage S32EX will hold all subsequent load, store and cache operation in the processor's R pipeline stage. Pipeline E-stage L32EX will hold all subsequent store in the processor's R pipeline stage. In the fast case, there will be no additional bubbles.

A.1.5 TIE Data- and Resource-Dependency Pipeline Bubbles

Instructions may be delayed in a pipeline for reasons other than operand dependencies. The most common situation is for two or more instructions to require a particular piece of the processor's hardware (called a "functional unit") to execute. If there are fewer copies of the unit than instructions that need to use the unit in a given cycle, the processor must delay some of the instructions to prevent the instructions from interfering with each other.

Modern processor pipeline design tends to avoid the use of functional units in varying pipeline stages by different instructions and to fully pipeline function-unit logic, which means that most instructions would conflict with each other over a shared functional unit only if they issue in the same cycle. Even so, there are usually a small number of cases in which a functional unit is used for several cycles.

Designer-defined extensions to the Xtensa processor can include shared functions that can be used by iterative instructions or by different instructions. An iterative operation uses a shared function multiple times across several cycles. Also, a shared function can be used by different instructions in different pipeline stages. This feature allows the designer to save area on a complex function, but it prevents multiple instructions from issuing in a pipelined fashion due to a resource dependency.

A.1.6 Pipeline Bubbles in SuperGather Gather/Scatter Configurations

For SuperGather gather/scatter configurations, if there is a gatherA instruction in the E pipeline stage followed directly by a gatherD instruction to the same GR (gather register) in the R pipeline stage, a one cycle bubble will be inserted between these two by holding gatherD in the R pipeline stage until the gatherA instruction moves out of the E pipeline stage. This is done because the gatherD instruction reads the GR result in the M pipeline stage and gatherA starts arbitrating for data RAM reads in the W+1 pipeline stage.

A scatterW instruction will hold in the R pipeline stage until the SuperGather engine is finished with the last write of all active scatter operations, including any that may be the CPU pipeline E-W stages. The scatterW instruction synchronizes non-SuperGather gather/scatter L/S operations with gather/scatter operations.

For SuperGather gather/scatter with ECC and read-modify-write configurations, store instructions will be held in the R pipeline stage until it is assured that any gatherD or scatter instruction will not cause a Global Stall. This is done because narrow store instructions need to do a read-modify-write and may put up a fence on a data RAM bank

access that a gatherA or scatter operation might be trying to use. This could cause a deadlock by blocking gatherA or scatter from accessing the data RAM. The gatherD instruction causing the global stall in the M pipeline stage depends on this and will not release until the operation finishes accesing the data RAM.

A.2 Branch Penalties

Xtensa processors incurs a branch-delay penalty for all instructions that redirect the instruction stream. Such instructions include branch instructions, jump instructions, call instructions, return instructions, and LOOP-type instructions that branch over the loop body. In all these cases, the Xtensa processor calculates the branch-target address in the pipeline's E stage, and that address goes to the instruction memory on the same cycle.

For a 5-stage pipeline, the result is a minimum 2-cycle branch-delay penalty as shown in Figure A–200. Because Xtensa processors fetch data in aligned groups of 4 or 8 bytes (depending on the configuration), the instruction at the target address can span two such groups and require two instruction-fetch cycles to fetch the complete instruction. In this case, the branch penalty increases to three cycles, as shown in Figure A–201. The Xtensa compiler (XCC) will often rearrange code to avoid branching, which eliminates the branch penalty. The Xtensa XCC compiler can also use wider versions of instructions (such as `addi` rather than `addi.n`) and will insert harmless nops to align branch targets so that the resulting branch penalty is two cycles rather than three.

For a 7-stage pipeline, the branch penalty is three cycles if the target instruction is contained within a fetch group, or four cycles if it spans two fetch groups, as shown in Figure A–202 and Figure A–203. The Xtensa XCC compiler will often rearrange code and align the target to get a branch penalty of three cycles rather than four.

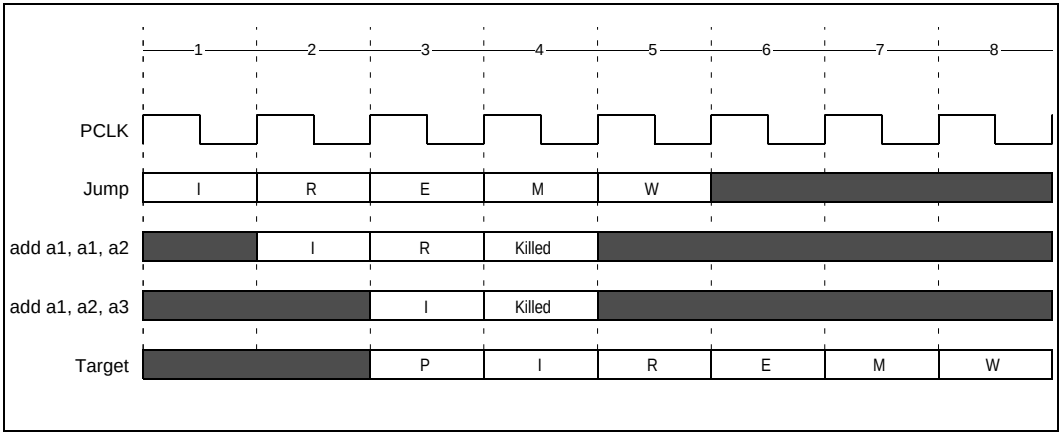


Figure A–200. 5-Stage Pipeline Branch-Delay Penalty with Aligned Target

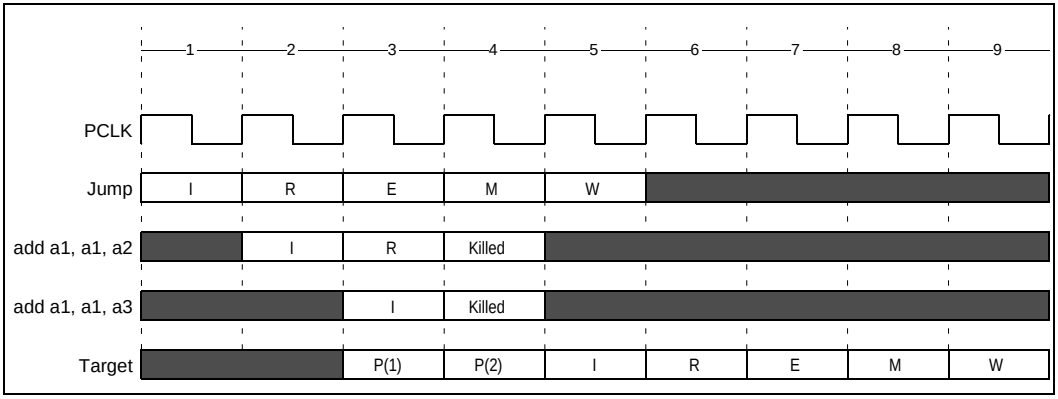


Figure A–201. 5-Stage Pipeline Branch-Delay Penalty with Unaligned Target

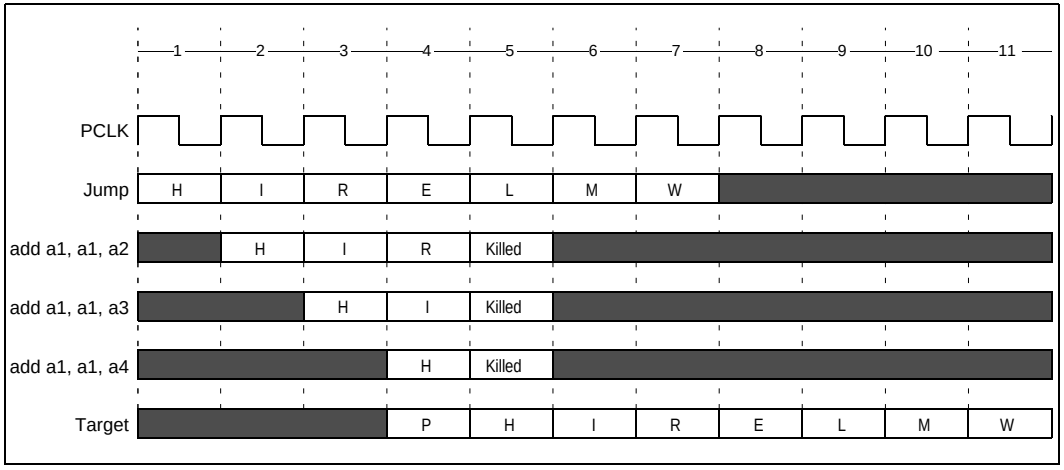


Figure A–202. 7-Stage Pipeline Branch-Delay Penalty with Aligned Target

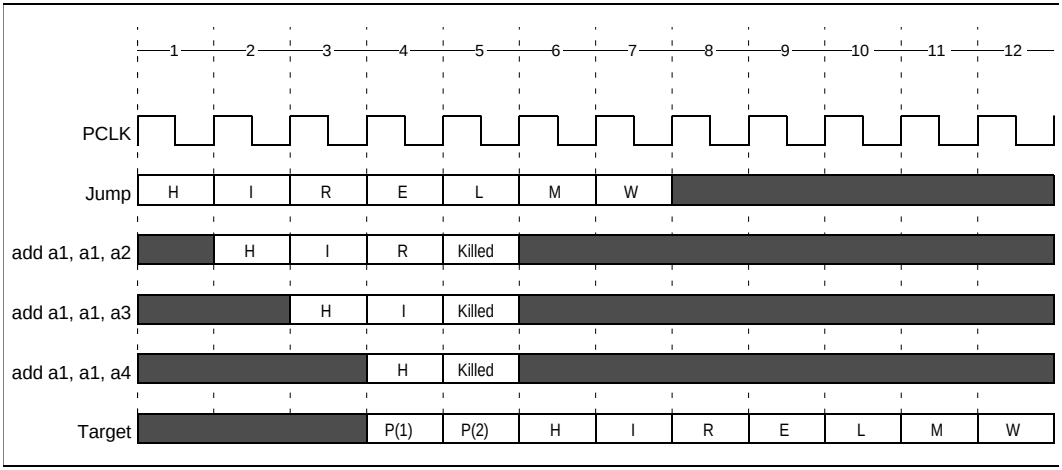


Figure A–203. 7-Stage Pipeline Branch-Delay Penalty with Unaligned Target

A.3 Loop Bubbles

The LOOP, LOOPGTZ, and LOOPNEZ instructions implement a zero-overhead loop operation, using the special registers LBEG (points at the beginning of the loop), LEND (points at the first instruction beyond the end of the loop body), and LCOUNT (counts the number of iterations). In the best case, there will be no bubbles in the pipeline due to branching back to LBEG except for a 1-cycle bubble at the end of the first iteration. A typical loop execution for a 5-stage pipeline and a 2-instruction loop is shown in Figure A–204.

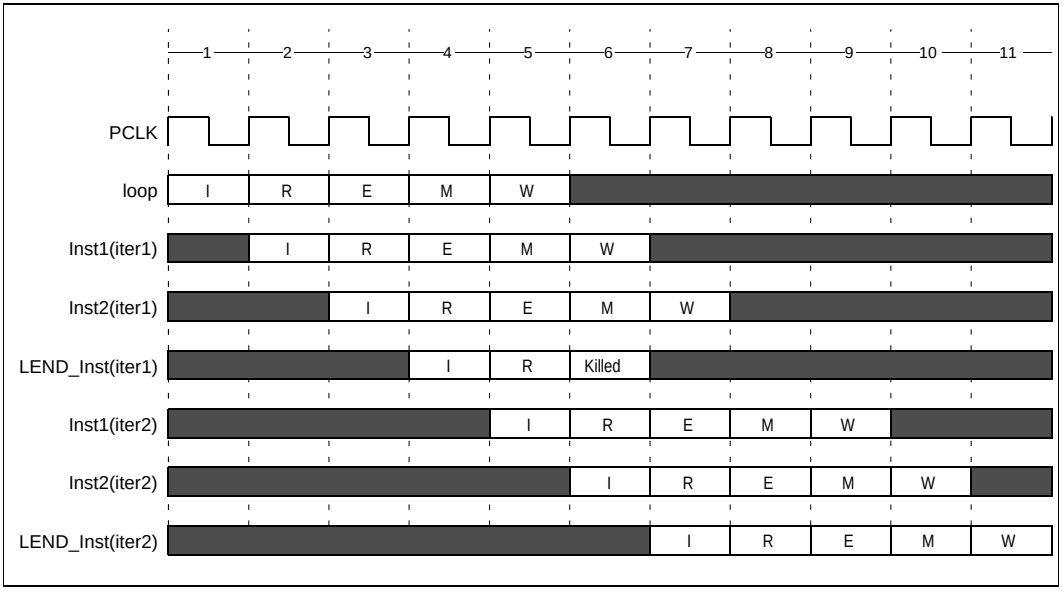


Figure A–204. Typical Execution of a 2-Instruction, 2-Iteration Loop on a 5-Stage Pipeline

There are a number of situations that cause bubbles in the pipeline during loop execution, especially for loops with a very small number of instructions in the loop body. These bubbles do not generally cause performance issues since they only occur on the very first or very last loop iteration.

Generally, having a minimum 2-instruction loop body for a 5-stage pipeline and a 3-instruction loop body for 7-stage pipeline minimizes these extra bubble situations.

Some applications may be so performance sensitive that a complete understanding of the situations that cause loop bubbles is required, including the bubbles on the first and last iterations of a loop. These bubbles are generally caused by the fetch engine using out-of-date values of LBEG, LEND, and LCOUNT. For instance, when a LOOP instruction enters the E stage, the portion of the pipeline that fetches instructions may have already made a fetch decision based on the old values of LBEG, LEND, and LCOUNT and corrective action must be taken to begin fetching based on the new values. All the situations that cause bubbles in the pipeline are detailed below.

Generally, the 7-stage pipeline is more complex and has more bubbles.

Again it should be emphasized that the loop-bubble cases generally cause minimal performance degradation. Consequently, only developers with applications that require every cycle to be counted need concern themselves with the following complex cases.

A.3.1 5-Stage Pipeline Loop Bubbles

The following cases cause pipeline bubbles during loop execution:

- 1-cycle bubble on the first loopback of a loop.
- 1-cycle bubble on the first loopback after an ISYNC instruction, a change in the LEND register, or an exception.
- 1-cycle bubble after the very last loop iteration of a 1-instruction loop. The bubble appears between the last instruction of the loop body and the instruction at LEND. There will not be an extra bubble if the last instruction in the loop body stalls in the R stage for some other reason. Note that this bubble can be eliminated by having at least a 2-instruction loop.

The following situations cause a pipeline replay to occur. The replay penalty of LBEG is five cycles in a 5-stage pipeline:

- Replay the first instruction following the LOOP, LOOPNEZ, or LOOPGTZ instruction if the instruction-fetch portion of the pipeline has fetched back to an old value of LBEG, based on old values of LBEG, LEND, and LCOUNT. After the instruction replays, LBEG, LEND, and LCOUNT will have new values, and instruction fetch can proceed correctly.
- Replay the first instruction after the LOOP, LOOPNEZ, or LOOPGTZ instruction for all 1-iteration loops that enter the loop with an old, non-zero LCOUNT value.

A.3.2 7-Stage Pipeline Loop Bubbles

The following cases cause pipeline bubbles during loop execution:

- 2-cycle bubble on the first loopback of a 1-instruction loop where the instruction does not stall in the R stage due to a resource dependency.
- 1-cycle bubble on the first loopback of a 2-instruction or larger loop, or a 1-instruction loop where the instruction is held in the R stage for one cycle due to a resource dependency.
- 1-cycle bubble on the first loopback after an ISYNC instruction, a change in the LEND register, or an exception.
- 2-cycle bubble after the last iteration of a 1-instruction loop where the instruction does not stall in the R stage due to a resource dependency. Note that this bubble can be eliminated by having at least a 3-instruction loop.
- 1-cycle bubble after the very last loop iteration of a 2-instruction loop, or a 1-instruction loop where the instruction is held in the R stage for one cycle in the last iteration. Note that this bubble can be eliminated by having at least a 3-instruction loop.
- 1-cycle bubble after the last iteration of a loop with more than one or two instructions, but a loop body that fits into one, two, or three aligned fetch widths (the instruction fetch width can be 4-, 8-, or 16-bytes depending on the type of wide in-

structions defined), if the instruction at LEND is itself not completely contained within an aligned instruction fetch width. This bubble occurs because the fetch portion of the pipeline fetches instructions in groups of aligned instruction fetch width bytes, and fetching the instruction at LEND in this case requires two cycles.

The following situations cause a pipeline replay to occur. The replay penalty of LBEG is seven cycles in a 7-stage pipeline:

- Replay the first instruction following the LOOP, LOOPNEZ, or LOOPGTZ instruction if the instruction-fetch portion of the pipeline has fetched back to an old value of LBEG, based on old values of LBEG, LEND, and LCOUNT. After the instruction replays, the LBEG, LEND, and LCOUNT will have new values, and the instruction fetch can proceed correctly.
- Replay the first instruction after the LOOP, LOOPNEZ, or LOOPGTZ instruction for all 1-iteration loops that enter the loop with an old, non-zero LCOUNT value.

In some situations, on the last instruction of the last iteration of a loop, the processor executes what amounts to a branch to the LEND instruction. These cases result in a branch-delay penalty of three or four cycles in a 7-stage pipeline:

- Branch to LEND to exit the loop whenever the processor holds the last instruction of the last loop iteration in the R stage for any reason.
- Branch to LEND to exit the loop in the case that the instruction fetch width is 64 or 128 bits, the loop body is three instructions long and fits entirely in one aligned fetch width, and the last byte of the last instruction in the loop body is not the last byte of the aligned fetch width.

A.4 Pipeline Replay Penalty and Conditions

Xtensa processors sometimes use a pipeline replay to simplify the implementation of certain functions or to recover from certain rare hazards that are not solved by pipeline stalls. A replay occurs in the W stage of an instruction, at which point the instruction in the W stage does not commit, and the instruction address is fed back to the instruction memory so that the instruction is re-executed. All instructions in the pipeline following the replayed instruction are killed. Figure A–205 shows the basic timing of an instruction replay for a 5-stage pipeline; note that the replay penalty is five cycles (six cycles if the instruction being replayed spans an aligned instruction fetch width).

The 7-stage case is similar; the replay penalty is seven cycles (eight cycles if the instruction being replayed spans an aligned instruction fetch width) for a 7-stage pipeline.

Replays can be caused by the instruction itself and its interaction with the rest of the processor, or in some cases it can be caused by the previous instruction. For instance, in all cases an ISYNC instruction causes the next instruction to replay so that this instruction is guaranteed to see the effects of all load, store, cache, TLB, and WSR instructions that affect instruction fetch and that preceded the ISYNC instruction.

In some cases, replays can be delayed. If an instruction-cache miss is in progress when the replay is first signaled, then the instruction-cache line fill is allowed to complete before the replay begins. A store to instruction RAM that is being blocked by `IRam`*n*`Busy` will also cause the replay to be delayed until the store to instruction RAM completes. If the external `RunStall` signal is asserted, the pipeline will be stalled and the replay will be delayed.

All the causes of replays are shown in Table A–166 and Table A–167. Table A–166 shows the cases that cause the instruction itself to be replayed, and Table A–167 shows the cases that replay the next instruction.

Note: The 7-stage penalty is two cycles longer in Figure A–205.

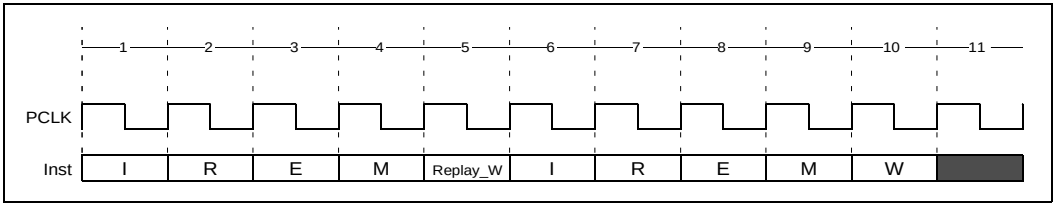


Figure A–205. 5-Stage Instruction-Replay Penalty

Table A–166. Conditions that Cause the Processor to Replay the Current Instruction

Condition	Description
Data load from PIF	A load from the PIF can be performed as part of a cache-miss refill, or an uncached or bypass load access.
Write-back-cache castout	A castout can be performed as part of a load or store instruction that evicts a dirty cache line or by a cache instruction such as DHWB.
Write-back instruction and store-tag buffer not empty	Any instruction that causes a write-back of the cache will be replayed if the store-tag buffer is not empty.
Load from instruction RAM or ROM	Any load from instruction RAM or instruction ROM except when using the L32R instruction in a processor without the full MMU with TLB option.
Load from instruction RAM or ROM with correctable memory error and DataExc bit of MESR register is not set	Any load from instruction RAM or instruction ROM that gets a correctable memory error with hardware error correction set will get an extra replay beyond the usual replay.
LICT instruction	Load from instruction cache Tag Memory.

Table A–166. Conditions that Cause the Processor to Replay the Current Instruction

Condition	Description
LICT instruction with correctable memory error and DataExc of MESR register is not set	Load from instruction cache Tag Memory that gets a correctable memory error with hardware error correction set will get an extra replay beyond the usual replay.
LICW instruction	Load from instruction cache Data Memory.
LICW instruction with correctable memory error and DataExc of MESR register is not set	Load from instruction cache data memory that gets a correctable memory error with hardware error correction set will get an extra replay beyond the usual replay.
IPFL Miss	IPFL instruction that misses in the instruction cache.
Load hits older stores in both load/store units (only for dual load/store configurations)	To maintain data coherence, the processor replays a younger load in the pipeline when it is loading from an address that matches two older stores (each initiated by one of the two load/store units) that are not yet written to memory or to the processor's write buffer. This replay does account for user-defined byte disables.
Correctable memory error occurs on either data cache, data tag or data RAM and the DataExc bit of the MESR register is not set	Any instruction that causes a correctable memory error with hardware error correction enabled will have an extra replay for the hardware to correct the data.
Unaligned load or store instruction that spans a loadstore width boundary	Any unaligned load or store instruction that crosses a loadstore width boundary with the unaligned handled by hardware option set will replay once to process the unaligned data.
S32C1I instruction that hits a dirty line and store-tag buffer not empty	An S32C1I instruction that requires a castout of the cache line will replay if the store tag buffer is not empty.
Load/Store address translation mismatch	The physical address of a load or store matches the Response Buffer physical address but the virtual address does not. This can occur if multiple virtual addresses map to the same physical address.
A cache miss to the same address that is currently being castout	During the rare case where a cache miss matches the same line that is being castout, the processor will replay.
Not enough write ids	Write through stores will replay where there aren't enough write ids available on the PIF
Prefetch replay	The prefetch control unit may replay the processor under various conditions in order to prevent refill conflicts in the pipeline.
Block prefetch replay	Block prefetch operations cause memory and cache instructions in the pipeline to replay to preserve ordering

Table A–167. Conditions that Cause the Replaying of the Next Instruction

Condition	Description
ISYNC	ISYNC instruction.
Store to instruction RAM	Any store instruction going to an instruction RAM.
SICT instruction	Store to instruction cache tag memory.
SICW instruction	Store to instruction cache data memory.
IHI,III instructions	Instruction-cache manipulation instructions IHI and III. (Instruction-Cache Hit Invalidate and Instruction-Cache Index Invalidate.)
IHU,IIU instructions	Instruction-cache manipulation instructions IHU and IIU. (instruction-cache Hit Unlock and instruction-cache Index Unlock)
IPFL Instruction	IPFL instruction that hits in the cache and succeeds in locking the cache line.
ITLB instructions	Any ITLB instruction: WITLB, IITLB, RITLB0, RITLB1, PITLB.
Special LOOP cases	Instruction following a LOOP type instruction is replayed if the fetch pipeline has based fetch decisions on incorrect values of LBEG,LEND, or LCOUNT. This can also occur in certain instruction bus error cases.
Exclusive Access Replay	If the L32EX (exclusive load) instruction has a partial address match with an inbound write, L32EX is replayed. If a store (regular or exclusive) has a partial address match with an inbound exclusive access, the store is replayed.

A.5 Instruction-Cache Miss Penalty

Instruction cache misses are signaled to the external interface, cache controller, or PIF controller in the R stage of an instruction that uses bytes from a reference that missed in the instruction cache. Depending on whether Early Restart is configured or not, the Instruction-Cache Miss Penalty will be different. If Early Restart is not configured, then the entire cache line response must be received and filled into the cache before the pipeline restarts. If Early Restart is configured, then the pipeline is restarted as soon as the critical data from the cache line has arrived.

A.5.1 No Early Restart Instruction Cache Miss

In the no Early Restart case, the instruction that takes the miss and all following instructions are killed and the pipeline waits for the instruction-cache line fill to complete before re-executing the instruction.

The instruction-cache line fill time is described by the following formula:

$ICache\_Fill\_Cycles = 1 + T_{arbitration} + T_{latency} + T_{datacycles} + T_{waitcycles}$

- $T_{arbitration} = T_{writebufferflush} + T_{loadrequestlatency}$
  - Minimum: 0 cycles
  - Maximum: load/store unit count (2) + writebuffer depth (1,2,4,8,16,32) + prefetch castout buffer depth (2*Cache line bytes / PIF width)

    Note: The number of load/store units determines the maximum number of outstanding load requests that are possible.
  - If the Prioritize Loads Before Store Operations configuration option has not been selected, then all pending operations in the write buffer will be completed and any pending load requests will be performed before the processor will issue the cache's instruction-fill request on the PIF. The time required to empty the write buffer time depends on the number of writes queued in the write buffer. Note that the write-buffer depth is configurable. In general, the write buffer attempts to complete all pending write operations whenever it is not empty. Therefore, performing all pending operations in the write-buffer only adds to the instruction-fetch latency if there are bus-wait cycles on the PIF. In the best case, finishing all pending operations in the write-buffer requires zero cycles.
  - If the Prioritize Loads Before Store Operations configuration option has been selected, the processor will issue the cache's instruction-fill requests before handling all pending operations queued in the write buffer, except for previously committed castouts in any of the write buffers.
  - If a prefetch to L1 is configured also, the prefetch castout buffer needs to be emptied before the processor will issue the cache's instruction-fill request on the PIF.
- $T_{latency}$
  - Minimum: 1 cycle
  - Maximum: system-dependent latency, unbounded
  - The latency over the PIF adds to instruction-miss latency.
- $T_{datacycles}$
  - (Cache line bytes / PIF width)-cycles
  - Number of cycles required to transfer all the cache-line data. This figure depends on the PIF width and the cache-line size. For instance, for a PIF width of 32-bits (4-bytes) and a 16-byte instruction-cache line width, four data cycles are required. The minimum number of cycles is one, corresponding to a minimum cache-line width of 16-bytes and a maximum PIF width of 128-bits (16-bytes).

- $T_{waitcycles}$
  - Minimum: 0 cycles
  - Maximum: system-dependent bandwidth, unbounded
  - Number of wait cycles on the PIF. Data cycles are not necessarily contiguous and can be separated by wait cycles or cycles used to service other requests.

The complete Instruction-Cache miss penalty can then be described by the following equation:

$$Instruction\_Cache\_Miss\_Penalty = T_{cache\_fill\_cycles} + T_{pipeline\_restart\_latency} + T_{alignment}$$

- $T_{cache\_fill\_cyles}$
  - Fill time described by the Cache_Fill_Cycles equation above
- $T_{pipeline\_restart\_latency}$
  - Three cycles for a 5-stage pipeline, four cycles for a 7-stage pipeline
- $T_{alignment}$
  - Minimum:
  - Maximum: 1 if the instruction is not completely contained in an aligned fetch width.

Figure A–206 shows an instruction-cache line-fill operation. Note that one extra cycle of latency will be incurred if the instruction is not completely contained within an aligned instruction fetch width (4, 8 or 16-bytes depending on the configuration) because the pipeline will be required to fetch the entire first instruction when it restarts after the cache-line fill.

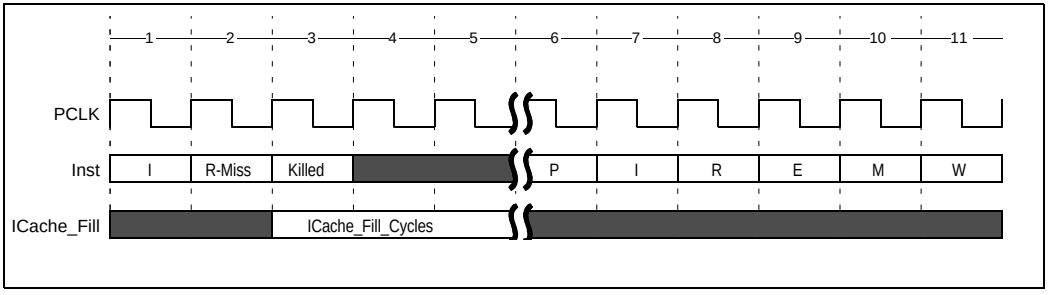


Figure A–206. Instruction-Cache Fill and Pipeline-Restart Latency, No Early Restart

A.5.2 Early Restart Instruction Cache Miss

In the Early Restart case, the instruction that takes the miss and all following instructions are killed and the pipeline immediately restarts and then stalls in the I-stage of the pipeline waiting for the critical data to be bypassed from the Response Buffer. The actual re-

fill of the cache line takes place in the background using free instruction cache line cycles. This means that the cache miss penalty depends on the latency of the cache miss request and response rather than on the complete cache miss fill cycles.

The instruction-cache line latency time is described by the following formula:

$ICache\_Latency\_Cycles = 1 + T_{arbitration} + T_{latency} + T_{datacycles} + T_{waitcycles}$

- $T_{arbitration} = T_{writebufferflush} + T_{loadrequestlatency}$
  - Minimum: 0 cycles
  - Maximum: load/store unit count (2) + writebuffer depth (1,2,4,8,16,32) + prefetch castout buffer depth (2*Cache line bytes / PIF width)
    Note: The number of load/store units determines the maximum number of outstanding load requests that are possible.
  - If the Prioritize Loads Before Store Operations configuration option has not been selected, then all pending operations in the write buffer will be completed and any pending load requests will be performed before the processor will issue the cache's instruction-fill request on the PIF. The time required to empty the write buffer time depends on the number of writes queued in the write buffer. Note that the write-buffer depth is configurable. In general, the write buffer attempts to complete all pending write operations whenever it is not empty. Therefore, performing all pending operations in the write-buffer only adds to the instruction-fetch latency if there are bus-wait cycles on the PIF. In the best case, finishing all pending operations in the write-buffer requires zero cycles.
  - If the Prioritize Loads Before Store Operations configuration option has been selected, the processor will issue the cache's instruction-fill requests before handling all pending operations queued in the write buffer, except for previously committed castouts in any of the write buffers.
  - If a prefetch to L1 is configured also, the prefetch castout buffer needs to be emptied before the processor will issue the cache's instruction-fill request on the PIF.
- $T_{latency}$
  - Minimum: 1 cycle
  - Maximum: system-dependent latency, unbounded
  - The latency over the PIF adds to instruction-miss latency.
- $T_{datacycles}$
  - Minimum: (Instruction Fetch Width/ PIF width)-cycles
  - Maximum: (Cache line bytes / PIF width)-cycles
  - Number of cycles required to the critical data over the PIF. If the Critical Word First configuration is chosen the critical data will always arrive first, and the minimum latency will be achieved. If the Critical Word First configuration option is

not chosen then the entire cache line may need to arrive. These numbers depends on the PIF width, the Instruction Fetch width, the cache-line size. For instance, for a PIF width of 32-bits (4-bytes), an Instruction Fetchwidth of 4-butes, and a 16-byte instruction-cache line width, 1 data cycle is required for the critical data, and four data cycles are required for the entire cache line.

- $T_{waitcycles}$
  - Minimum: 0 cycles
  - Maximum: system-dependent bandwidth, unbounded
  - Number of wait cycles on the PIF. Data cycles are not necessarily contiguous and can be separated by wait cycles or cycles used to service other requests.

The complete Instruction-Cache miss penalty can then be described by the following equation:

$Instruction\_Cache\_Miss\_Penalty = T_{cache\_latency\_cycles} + T_{alignment}$

- $T_{cache\_latency\_cyles}$
  - Fill time described by the ICache_Latency_Cycles equation above
- $T_{alignment}$
  - Minimum: 0
  - Maximum: 1 if the instruction is not completely contained in an aligned fetch width.

Figure A–207 shows an instruction-cache line miss operation in the early restart case.

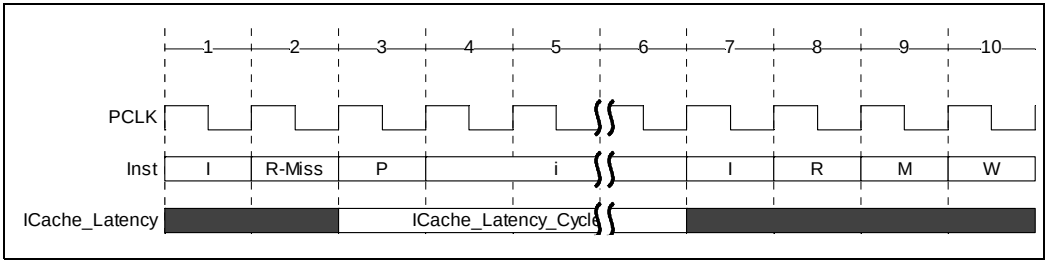


Figure A–207. Instruction-Cache Fill and Pipeline-Restart Latency, Early Restart

A.5.3 Uncached Instruction Execution Latency

It is also possible to execute instructions in an uncached manner where all instruction bytes are fetched over the PIF. Because executing uncached code is very slow, uncached code is typically executed only at startup time for a basic reset routine that initializes the instruction-cache tags and other state essential to start executing code out of the cache.

An uncached instruction fetch will fetch bytes in aligned instruction fetch width (4, 8 or 16 bytes depending on the configuration) and saves this data in a temporary holding buffer. If the processor is executing instructions sequentially and an instruction spans two aligned fetch widths, then another uncached fetch will occur, and the instruction will be formed from a combination of the data in the temporary holding register and the newly fetched data. If the processor branches to an instruction that spans two fetch widths, then two uncached fetches will occur before the instruction can be executed.

The formula describing the number of cycles required to fetch uncached instructions over the PIF is the same as for fetching a complete cache line in the no early restart case (see Appendix A.5.1 “No Early Restart Instruction Cache Miss”), except that $T_{data\text{-}cycles}$ is always 1. Figure A–208 shows an instruction executing uncached. Note that the pipeline requires one less cycle to restart in this case (because no P stage is required). In the case that the processor branches or jumps to an unaligned instruction, two uncached fetches will be necessary before the instruction can execute.

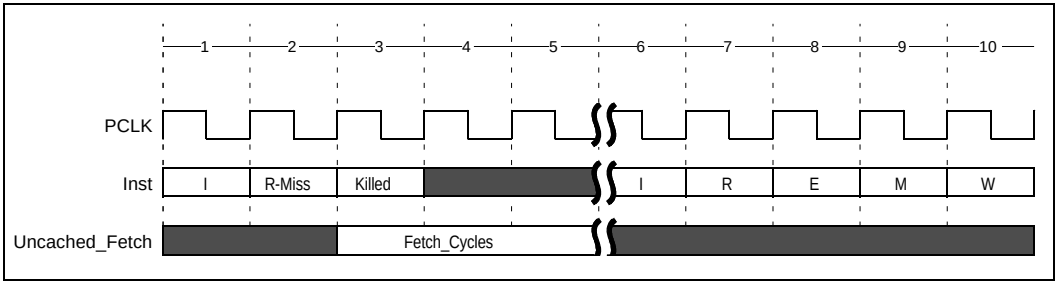


Figure A–208. Uncached Instruction Fetch and Pipeline-Restart Latency

A.6 Data-Cache Miss Penalty

A.6.1 No Early Restart Cacheable Miss

If loads or stores cause a data-cache miss and the missing line is allocatable, it must be brought from system memory (over the PIF) and loaded into the cache. Misses are detected in the M stage of the load or store, whereupon the instruction is killed and the fill

sequence initiated. The pipeline waits for the fill to complete before allowing the re-executed instruction to read the data cache again, inserting R-stage bubbles if necessary.

The data-cache line miss penalty is described by the following formula:

Cache_Miss_Penalty =

$$1 + T_{arbitration} + T_{castoutread} + T_{latency} + T_{datacycles} + T_{waitcycles} + T_{replay} + T_{inststraddle}$$

$$T_{arbitration} = T_{storebufferflush} + T_{writebufferflush}$$

- Minimum: 1
- Maximum: store buffer depth (2,3) + busy cycles + writebuffer depth (1,2,4,8,16,32) + wait cycles
- If the Prioritize Loads Before Store Operations configuration option has not been selected, then all pending operations in the write buffer will be completed and any pending load requests will be performed before the processor will issue the cache's data-fill request on the PIF. The time required to complete all pending operations in the store buffer and write buffer depends on the number of writes that have been queued and the availability of the write targets. The store-buffer depth is two for a 5-stage pipeline and three for a 7-stage pipeline.

  The write-buffer depth is configurable from 1 to 32 entries. Generally, the store buffer and write buffer attempt to complete all pending operations whenever they are not empty, but a Busy assertion on local-memory interfaces or an occupied PIF can indefinitely extend the time before the cache's data-fill request can go out on the PIF. Depending on how the addresses in the cache's data-fill requests compare to the addresses of the queued operations pending in the processor's store and write buffers, the pending store- and write-buffer operations may be completely or partially executed before the cache's request is issued over the PIF.
- If the Prioritize Loads Before Store Operations configuration option has been selected, the processor will issue the cache's fill requests before handling all pending operations queued in the write buffer, assuming there are no address conflicts between the cache-line fill transaction and the pending stores in the write buffer.

$T_{castoutread} + T_{tagbufferflush}$

- Minimum: 0
- Maximum: 1 or 2 (for a 5- or 7-stage pipeline, respectively) to read the cache for the castout data + (cache line bytes / PIF width) - 1 - (overlap with $T_{latency}$ and $T_{waitcycles}$)
- After the data-fill request has gone out on the PIF, a castout of the victim line is initiated—if a write-back cache is configured. These two transactions can complete independently on the fully bidirectional PIF, but they compete for access to the single-port cache. Even if the fill-response data has arrived at the processor over the PIF, it

is not written to the cache until the castout data has been read out of the cache. The number of cycles required for the castout read is equal to the number of cycles required to determine whether a castout is required or not, plus the time of the read itself, which is based on the cache-line-size-to-PIF-width ratio. If a castout is required, these two transactions overlap by a cycle. This overlap is in addition to the overlap of the entire castout read time with the fill-response arrival time.

$T_{latency}$

- Minimum: 1 cycle
- Maximum: system-dependent latency, unbounded
- The latency over the PIF adds to data-miss latency. This is the latency to the arrival of the first PIF-width data transfer.

$T_{datacycles}$

- (Cache line bytes / PIF width)-cycles
- Number of cycles required to transfer the cache-line data. This number depends on the PIF width and the cache-line size. For instance, for a PIF width of 32-bits (4-bytes) and a 16-byte data-cache line width, four data-transfer cycles are required. The minimum number of cycles is 1, corresponding to a minimum cache-line width of 16-bytes and a maximum PIF width of 128-bits (16-bytes).

$T_{waitcycles}$

- Minimum: 0 cycles
- Maximum: system-dependent bandwidth, unbounded
- Number of wait cycles on the PIF. Data cycles are not necessarily contiguous and can be separated by wait cycles or by cycles used to service other requests.

T_{replay}

- Fixed number of cycles: two cycles for a 5-stage pipeline and three cycles for a 7-stage pipeline.
- Number of cycles from the pipeline's R stage to its M stage.

$T_{inststraddle}$

- Minimum: 0 cycles

- Maximum: 1 cycle; the extra cycle of latency is added when the replayed instruction straddles two aligned fetch widths, therefore requiring two accesses to fetch the whole instruction, and when either of the following is true:
  - The instruction caused a cache miss on a 7-stage pipeline where $T_{datacycles} = 1$, or
  - The instruction is an uncached load, and IRAM or ICache is configured with either ECC or Parity protection options, regardless of the fetch address of the load instruction.

Figure A–209 shows the best-case data-cache line-fill operation.

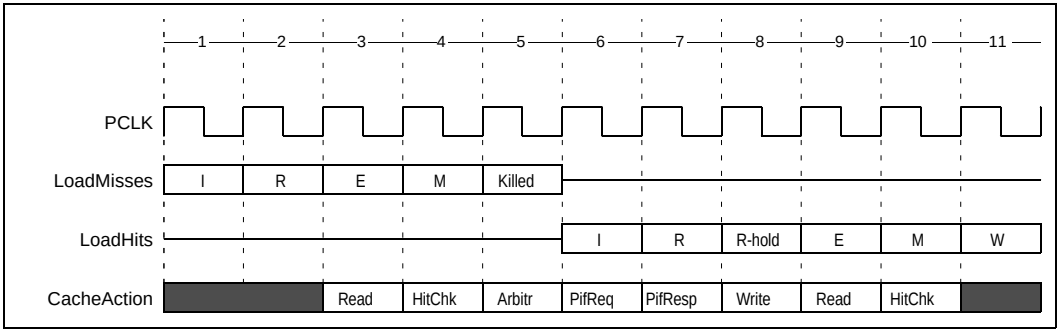


Figure A–209. Example of Data Cache Fill

A.6.2 Early Restart Cacheable Miss

In the Early Restart case when a load instruction takes a miss, all following instructions are killed and the pipeline immediately restarts, stalling in the M-stage of the pipeline until the critical data is bypassed from the Response Buffer. The actual refill and possible castout of the cache line takes place in the background using free data cache cycles. This means that the cache miss penalty for load instructions depends on the latency of the cache miss request and response rather than on the complete cache miss fill cycles.

Store instructions behave differently. When a store instruction takes a miss, all following instructions are killed and the instruction stalls in the M-stage of the pipeline until the entire line is filled. As a result, the miss penalty for store instructions depends on the latency of the entire cache miss fill.

If loads or stores cause a data-cache miss and the missing line is allocatable, it must be brought from system memory (over the PIF) and loaded into the cache. Misses are detected in the M stage of the load or store, whereupon the instruction is killed and the fill sequence initiated.

The pipeline waits for the critical word to arrive before allowing the re-executed instruction to commit, stalling in the M stage if necessary.

The data-cache line miss penalty is described by the following formula:

Cache_Miss_Penalty =

$1 + T_{arbitration} + T_{latency} + T_{waitcycles} + T_{replay} + T_{inststraddle}$

$T_{arbitration} = T_{storebufferflush} + T_{writebufferflush} + T_{tagbufferflush}$

- Minimum: 1
- Maximum: store buffer depth (2,3) + busy cycles + writebuffer depth (1,2,4,8,16,32) + wait cycles
- If the Prioritize Loads Before Store Operations configuration option has not been selected, then all pending operations in the write buffer will be completed and any pending load requests will be performed before the processor will issue the cache's data-fill request on the PIF. The time required to complete all pending operations in the store buffer and write buffer depends on the number of writes that have been queued and the availability of the write targets. The store-buffer depth is two for a 5-stage pipeline and three for a 7-stage pipeline.

  The write-buffer depth is configurable from 1 to 32 entries. Generally, the store buffer and write buffer attempt to complete all pending operations whenever they are not empty, but a Busy assertion on local-memory interfaces or an occupied PIF can indefinitely extend the time before the cache's data-fill request can go out on the PIF. Depending on how the addresses in the cache's data-fill requests compare to the addresses of the queued operations pending in the processor's store and write buffers, the pending store- and write-buffer operations may be completely or partially executed before the cache's request is issued over the PIF.
- The store tag buffer is flushed in parallel with the store buffer and can be up to 4 entries deep.
- If the Prioritize Loads Before Store Operations configuration option has been selected, the processor will issue the cache's fill requests before handling all pending operations queued in the write buffer, assuming there are no address conflicts between the cache-line fill transaction and the pending stores in the write buffer.

$T_{latency}$

- Minimum: 1 cycle
- Maximum: system-dependent latency, unbounded
- The latency over the PIF adds to data-miss latency. This is the latency to the arrival of the first PIF-width data transfer.

$T_{waitcycles}$

- Minimum: 0 cycles
- Maximum: system-dependent bandwidth, unbounded

- Number of wait cycles on the PIF. Data cycles are not necessarily contiguous and can be separated by wait cycles or by cycles used to service other requests.

T_{replay}

- Fixed number of cycles: two cycles for a 5-stage pipeline and three cycles for a 7-stage pipeline.
- Number of cycles from the pipeline's R stage to its M stage.

$T_{inststraddle}$

- Minimum: 0 cycles
- Maximum: 1 cycle; the extra cycle of latency is added when the replayed instruction straddles two aligned fetch widths, therefore requiring two accesses to fetch the whole instruction, and when either of the following is true:
  a. The instruction caused a cache miss on a 7-stage pipeline where $T_{datacycles} = 1$, or
  b. The instruction is an uncached load, and IRAM or ICache is configured with either ECC or Parity protection options, regardless of the fetch address of the load instruction.

Figure A–210 shows the best-case data-cache line-fill operation with early restart.

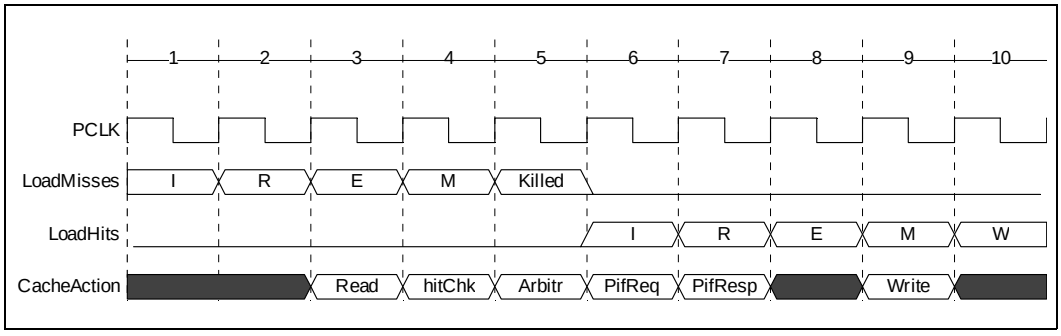


Figure A–210. Example of Data Cache Fill with Early Restart

A.6.3 Uncached Load

If the cache miss is caused by an attempted access to data that is not allocatable or in uncached or cache bypass region, no cache refill occurs. If the missing instruction is a store, the write is queued in the write buffer and the pipeline simply continues. If the missing instruction is a load, however, a replay still occurs to allow the data to be retrieved from the system memory or I/O. The pipeline waits for the data to be available in an uncached buffer before allowing the re-executed instruction to proceed, inserting R-stage bubbles if necessary.

The uncached-load penalty is described by the following formula:

Uncached_Load_Penalty = 1+ $T_{arbitration}$ + $T_{latency}$ + $T_{datacycles}$ + T_{replay} + $T_{inststraddle}$ + $T_{memoryerrors}$

- $T_{memoryerrors}$
  - Minimum: 0
  - Maximum: Two cycles for 5-stage pipelines and three cycles for 7-stage pipelines when IRAM or ICache is configured with either ECC or Parity protection options, regardless of the fetch address of the load instruction.

Figure A–211 shows an example of an uncached load for a 7-stage pipeline and instruction memory error protection.

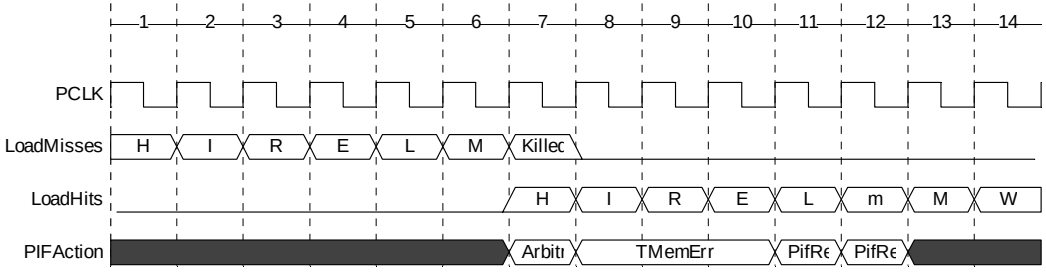


Figure A–211. Uncached Load for 7-stage Pipeline with I-Side ECC or Parity Protection

A.7 Exception Latency

When an exception occurs on a particular instruction, it is signaled in the W stage of that instruction. In the normal case, the address for the exception handler is output to the instruction memories on that same cycle. The first instruction of the exception handler is in the W stage five cycles (six cycles if the instruction being replayed spans an aligned instruction fetch width) later for a 5-stage pipeline and seven cycles (eight cycles if the instruction being replayed spans an aligned instruction fetch width) later for a 7-stage pipeline, as illustrated in Figure A–212.

A number of conditions can increase exception latency. The minimum exception latency assumes that there are no instruction-cache misses or other delays when fetching the exception handler instructions (for example, `IRam`*n*`Busy` when the interrupt handler is contained in instruction RAM, `RunStall` being active, etc.). If there are instruction-cache misses or delays, then the exception latency increases by the amount of time required to fill the instruction-cache line or by the number of cycles that the pipeline cannot progress due to instruction-stall conditions. If there is an instruction-cache line fill in progress when the exception is signaled, the exception latency increases by one cycle so that the instruction-cache line fill can be cancelled.

Note that a 7-stage case is similar to Figure A–212, but adds two extra cycles to the latency. The minimum latency assumes no instruction-fetch delays on the execution of the exception handler.

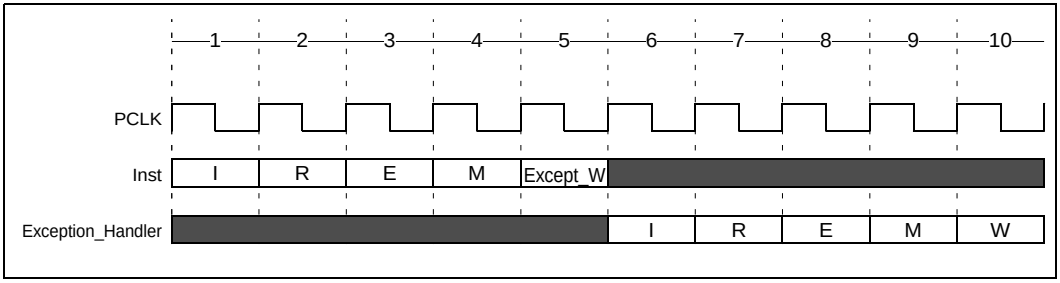


Figure A–212. 5-Stage Pipeline Minimum Exception Latency

A.8 Interrupt Latency

Interrupt latency is similar to exception latency except that interrupts are not associated with a particular instruction being in the W stage.

The minimum interrupt latency from the time an external interrupt signal becomes active at the processor input to the time that the first instruction of the interrupt handler reaches the W stage is seven cycles for the 5-stage pipeline, as shown in Figure A–213. This figure increases by two cycles for a 7-stage pipeline.

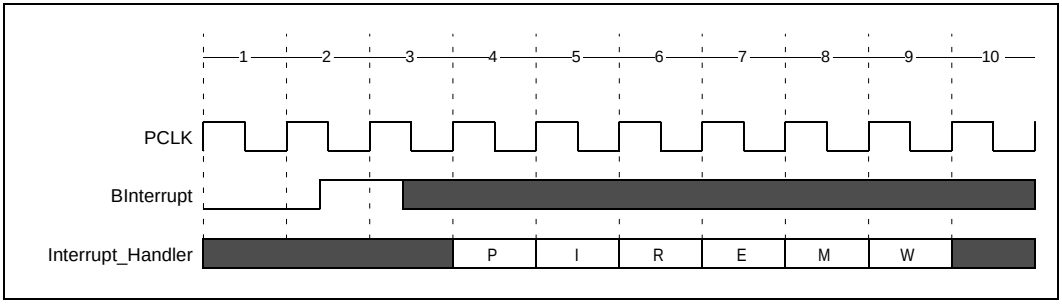


Figure A–213. 5-Stage Pipeline Minimum Interrupt Latency

In general, interrupt latency can be described as:

$T_{interrupt\_latency} = 2 + \text{pipeline depth} + T_{mask} + T_{instruction\_fetch\_latency}$

- Where T_{mask} is the time that an Interrupt is delayed due to interrupt delaying or masking conditions, and $T_{instruction\_fetch\_latency}$ is the latency of fetching the first instruction of the Interrupt Handler.

- $T_{mask} = T_{globalstall} + T_{bypass} + T_{castout} + T_{ocdhaltmode} + T_{autorefill} + T_{misc}$
  - Minimum: 0 cycles
  - Maximum: A sum of the following conditions. A number of conditions can increase the interrupt latency. In addition to interrupt masking by `INTENABLE` and `PS.INTLEVEL`, interrupts are masked under the following conditions, adding directly to interrupt latency.
  - Most `GlobalStall` cases (Section A.9) are interruptible and add 1-cycle of interrupt latency, which is necessary to turn off the stalling condition.
  - Global stall's due to the `RunStall` control are not interruptible.
  - Global stalls due to an instruction RAM busy are not interruptible.
  - Continuous Inbound DMA requests to instruction RAM can cause a global stall that is not interruptible.
  - All interrupts are masked during bypass data accesses (except those marked interruptible in the MPU) and read-conditional-write requests to the PIF. In fact, bypass and read-condition-write accesses are allowed to commit before interrupts are taken because they may have had side effects such as reading from a queue, in which case data would be lost if the read was interrupted. Please note however, that unaligned bypass data accesses that span a loadstore width with the Unaligned Handled By Hardware option configured are not atomic transactions and will not mask interrupts. Therefore, it is essential that no unaligned bypass transactions access devices with read side effects.
  - In configurations with a full MMU, interrupts are masked during hardware TLB refill operations.
  - Interrupts are delayed one cycle if they arrive during cache accesses, including castouts and cache refills.
  - The commit stage of a `WAITI` instruction may add one cycle.
  - Interrupts are delayed for two cycles after a store operation to instruction RAM reaches the pipeline's W stage, in addition to any stall cycles caused by the assertion of an instruction RAM's `Busy`.
  - Interrupts are delayed for one cycle following an instruction that causes the processor to replay the subsequent instruction (see Table A–167).
  - Interrupts are not taken while the processor is in OCD Halt Mode. The length time that the processor is in OCD halt mode is system-dependent.
- $T_{instruction\_fetch\_latency} = T_{delay+}T_{instruction\_miss\_latency}$
  - Minimum: 0 cycles
  - Maximum: Depends on various conditions described below
  - The fetch latency directly adds to the interrupt latency. In the fastest case, no extra cycles are added to the minimum interrupt latency due to instruction fetch delays.

- T_{delay}
  - Instruction fetches from local memories can be delayed by memory Busy's. The minimum additional latency from a local memory fetch is zero cycles. The maximum delay due to a busy memory is system-dependent and unbounded.
  - If the interrupt handler requires a fetch over the PIF (either a cache line miss or an uncached fetch), and a previous instruction miss is outstanding, then all the data must be received before the new cache miss can begin.
- $T_{instruction\_miss\_latency}$
  - If the first instruction of the Interrupt Handler takes an instruction cache miss, or needs to fetch instructions uncached over the PIF, then the Interrupt latency will be increased by the time to fetch this first instruction. See Section A.5 for more details on instruction cache miss delays.

The maximum interrupt latency depends heavily on the memory organization, location of vectors, global stalls, and bus waits. Some of these delays are simply additive. Other delays are absorbed in parallel so that the maximum delay of the group adds to the interrupt latency. Table A–168 summarizes the range of delays that may be seen when there are no external stall or busy conditions, with the only latency being due to the pipeline, instruction cache miss and an outstanding bypass load.

Table A–168. Interrupt Latency without External Stall of Busy Conditions

Vector location	Minimum Latency	Maximum Latency
Local memory or cache hit	7 cycles	7 + (Tbypass) cycles
Cache miss	11 cycles[1] 14 cycles[2]	11 + (Tbypass) cycles[1] 14 + (Tbypass) cycles[2]

NOTE: Latencies are for a 5-stage pipeline.

Add two latencies for 7-stage pipelines.

1. Assumes single-cycle cache refill, *i.e.*, cache line size == pif width

2. Assumes 4-cycle cache refill, i.e.,. cache line size == 4pif widths

A.9 GlobalStall Causes

A GlobalStall condition stalls the entire pipeline by gating the main clock to the pipeline. Some global stalls, such as stalls caused by assertion of a memory-busy handshake signal, are functionally necessary for the correct processor operation. Other stalls, such as stalls during cache-miss refills, occur even while the processor is idle to reduce power consumption. Table A–169 details the conditions under which a global stall will occur.

Note that in-progress external transactions (that were not themselves the cause of the global stall—that is, loads or stores to local memory or PIF, cache-line fills and castouts, TIE-queue transfers, and inbound-PIF requests to instruction or data RAM will all attempt to complete during a global stall.

Table A–169. Conditions that Cause GlobalStall

Condition	Description
Runstall	When asserted, external signal RunStall forces a pipeline stall (see Section 22.6 for details).
External IRamnBusy or IRomnBusy	If the processor is executing out of instruction RAM or instruction ROM and the corresponding Busy signal is active, the pipeline will stall except under certain conditions where it has filled its internal holding buffers enough to continue executing[1]. Similarly, if the processor is loading from instruction RAM or instruction ROM or storing to instruction RAM and the corresponding Busy signal is active, then GlobalStall will be active (see Section 18.3.2 for details).[2]
Instruction-cache line fills	The pipeline will stall while waiting for instruction-cache line data responses. The stall depends on whether Early Restart is configured or not (unstall after the critical data arrives of after the fill completes).
Data-cache line fills	The pipeline will stall while waiting for data-cache line fills to complete once the instruction causing the miss has been replayed and made it back to the pipeline's R stage for non early restart configurations. When early restart is configured, the pipeline will stall in the M stage until the critical word arrives.
Store buffers full	Where there's a store in the M stage and there are no store buffers available, the pipeline will stall.
Store matches store buffer of a different load store element	When a store in the M stage matches a store buffer of a different loadstore element, the processor will stall until the overlapping store buffer has dispatched
Prefetch stall	Under certain circumstances, a prefetch refill may stall the pipeline
Bank conflicts across load/store units when accessing data cache or data RAM.	When two loads in the same FLIX packet try to access the same bank of the same memory, the processor will stall until both load accesses have occurred sequentially.
Inbound PIF to instruction RAM	Inbound-PIF requests to an instruction RAM will cause an internal IRamBusy for the instruction RAM every cycle that the instruction RAM is being used by the inbound-PIF request. GlobalStall will occur as if there were an external IRamBusy for that instruction RAM.
TIE Queue blocking read	TIE queue blocking reads cause a GlobalStall if the external queue is empty.
TIE Queue blocking write	TIE queue blocking writes cause a GlobalStall if the external queue is full and there are 3 or more pending output-queue writes.
TIE Lookup not ready	TIE lookup's can be configured with an optional arbitration signal. A lookup request when the external arbiter is not ready can cause the pipeline to stall.

Table A–169. Conditions that Cause GlobalStall (continued)

Condition	Description
External DPortnBusym, DRamnBusym, or DRomnBusym	If the processor is attempting a load to the local data memory and the corresponding `Busy` signal is active, the pipeline will stall. Similarly, the pipeline will also stall if the processor is attempting a store to the local data memory and the corresponding `Busy` signal is active and its store buffer is full.
Inbound PIF to data RAM	Inbound-PIF requests to the data local memories will not cause any disruption to the normal pipeline execution except to maintain the target bandwidth requirements specified by the request. In such case, an internal Busy for the target local memory is asserted and, if the pipeline is also attempting to access the same local memory, GlobalStall will occur.
S32C1I instruction in the E stage immediately following an S32C1I inbound-PIF transaction	The Lock signal needs to be deasserted for at least one cycle in between S32C1I transactions to preserve atomicity. As a result, any S32C1I instruction immediately following an S32C1I inbound PIF transaction will be stalled for a single cycle.
Iterative Integer Multiply	Integer multiplication, when the iterative option is selected, is scheduled as a single-cycle latency operation, but causes the pipeline to stall for up to 5 cycles. A fully pipelined version is also available.
Integer Divide	Integer division requires 1 to 11 stall cycles for the quotient operation, and 2 to 12 stall cycles the remainder option. The operations are scheduled as single-cycle latency.
Non-pipeline read-modify-write for 32-bit data RAM error protection	When a high-priority read-modify-write operation of a non-pipeline agent, such as inbound PIF or iDMA is denied, that agent can cause a pipeline to stall to allow the read-modify-write operation to make progress.
SuperGather Gather/Scatter instructions	A gatherD instruction will cause a global stall in the M pipeline stage if there is a gatherA instruction to the same gather register active doing reads to the data RAM. It will release after the last gatherA read completes. A scatter instruction will cause a global stall in the M pipeline stage if it is determined that there are not enough entries in the scatter register FIFO for this new one. It will release when an entry frees up.

Notes:

1. Exact conditions under which no GlobalStall will occur when executing out of IRAM or IROM with corresponding Busy active are complex.Busy signals being active should be minimized when the processor is executing out of that memory.
2. A store to IRAM that has its Busy asserted effectively causes the pipeline to stall, but does not use the GlobalStall mechanism.

B. Notes on Connecting Memory to the Xtensa Core

This Appendix provides notes on the functional characteristics of memory connected to the Xtensa processor core. The general characteristics are described here, and they apply to any of the instruction or data caches, tag RAM, or other RAMs connected to the core. Only the read transaction descriptions apply to the instruction or data ROMs. If there is memory connected to the XLMI port, it must also exhibit the described behavior.

B.1 Memory Latency

Latency is the main factor that defines the memory's functional behavior. Latency describes the number of cycles between the Xtensa processor's assertion of the memory port's read address and enable, and the processor's acceptance of the read data from the memory. The latency number is always an integer, because it is only a functional measure and is decoupled from the AC timing of the signals to and from the memory. Latency is always referred to with respect to PCLK.

Only two memory latencies are possible for the Xtensa LX7 core: one or two cycles. The 5-stage pipeline requires all local memories to have a 1-cycle latency. Memory with a 1-cycle latency is also referred to as *flowthrough memory* by certain memory-compiler vendors. The 7-stage pipeline requires all local memories to have a 2-cycle latency. Memory with a 2-cycle latency is also referred to as *pipelined memory.*

Note: For Xtensa processors with memories, the staging flops are included as part of the synthesis flow.

B.2 1-Cycle Latency

Figure B–214 shows a conceptual picture of memory with 1-cycle latency.

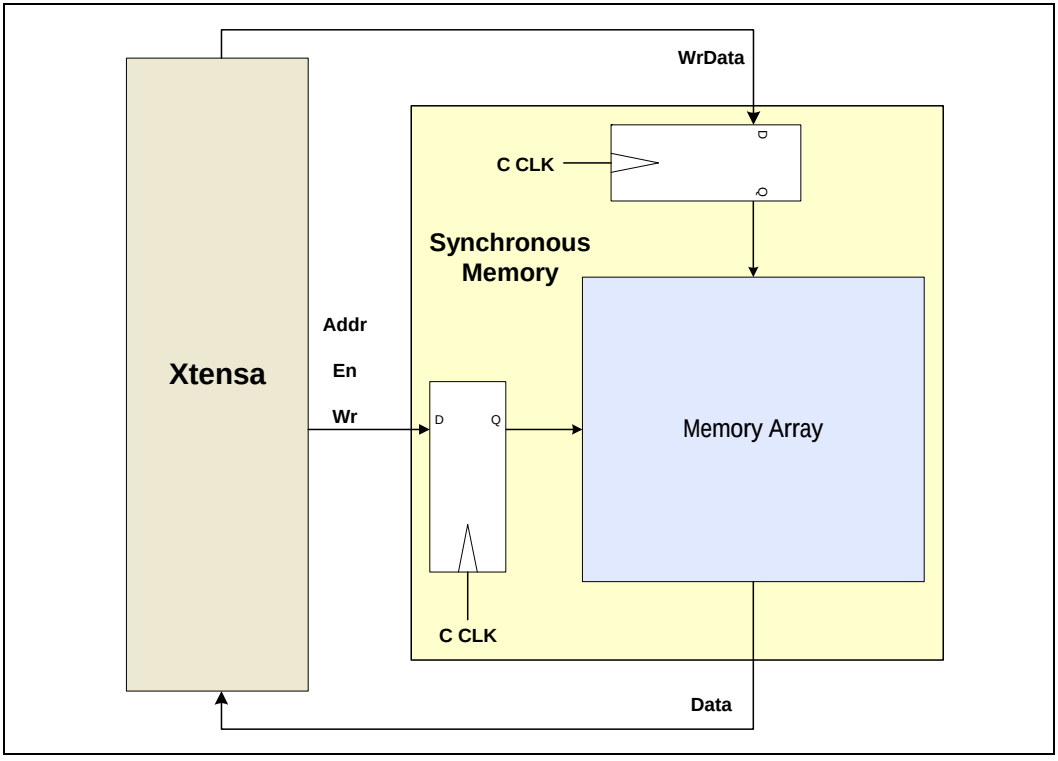


Figure B–214. Single-Port Memory with 1-Cycle Latency

The simplest memory read is shown in Figure B–215. Note that the Xtensa processor can negate the memory enable, En, before the clock edge when the processor samples the read data from the memory.

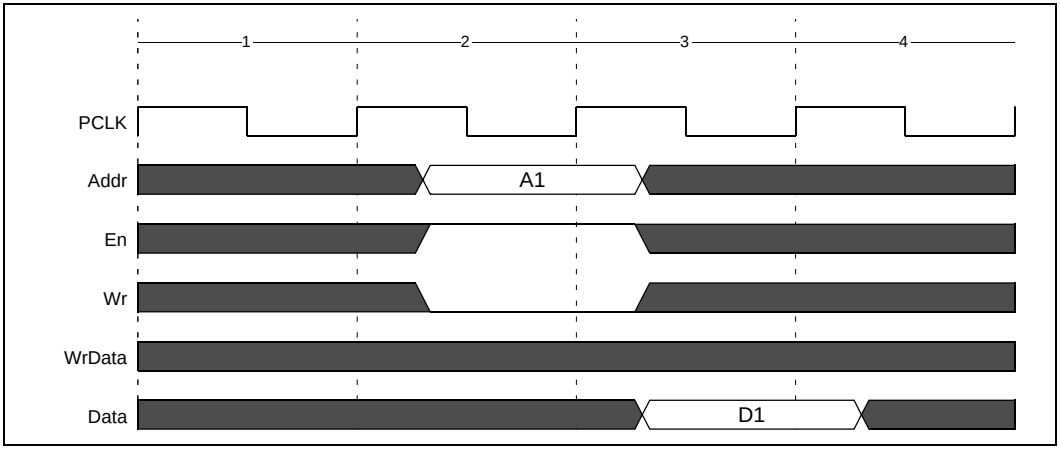


Figure B–215. Read from Memory with 1-Cycle Latency

Alternatively, En could be asserted for the next read. Figure B–216 shows back-to-back reads for memory with 1-cycle latency.

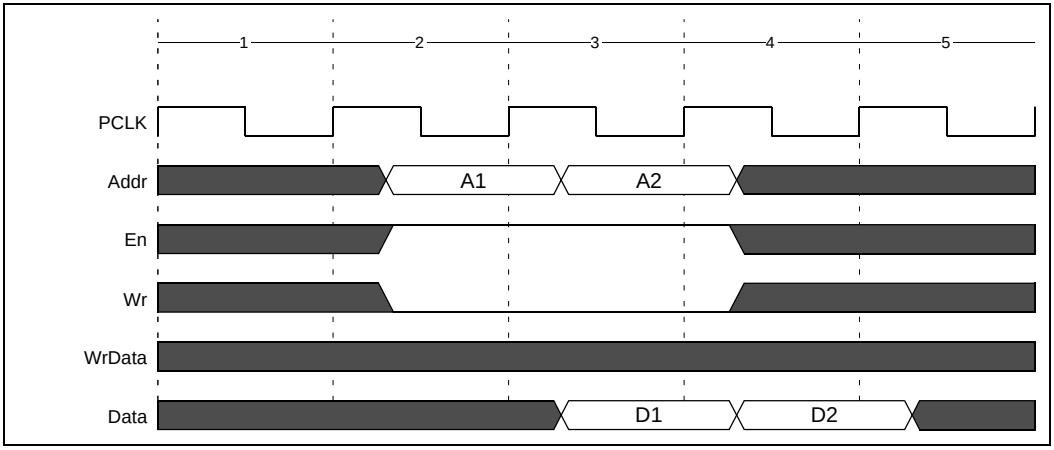


Figure B–216. Back-to-Back Reads from Memory with 1-Cycle Latency

Figure B–217 shows the simplest write for memory with 1-cycle latency.

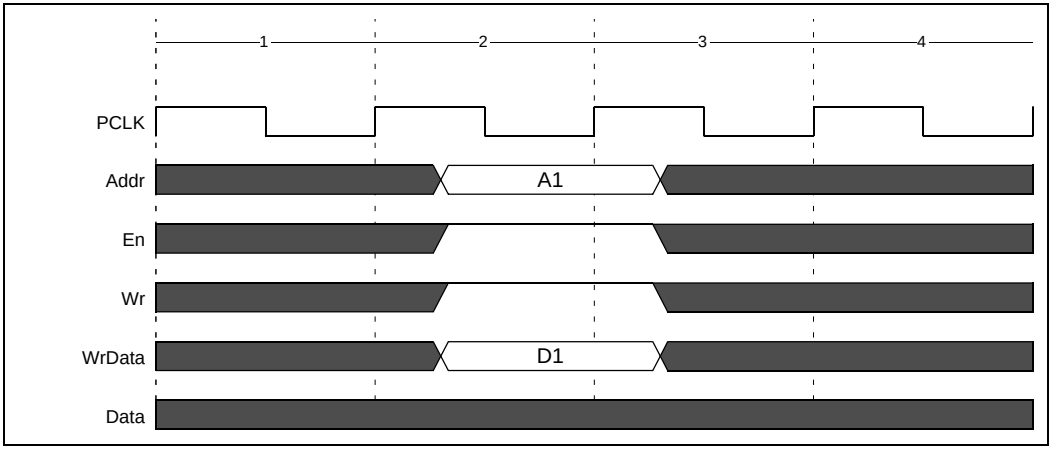


Figure B–217. Write to Memory with 1-Cycle Latency

The precise moment of the real update to the memory cell will depend on the memory design, but the Xtensa processor expects that a read of the same address in the cycle immediately following a write will yield the new data, as shown in Figure B–218.

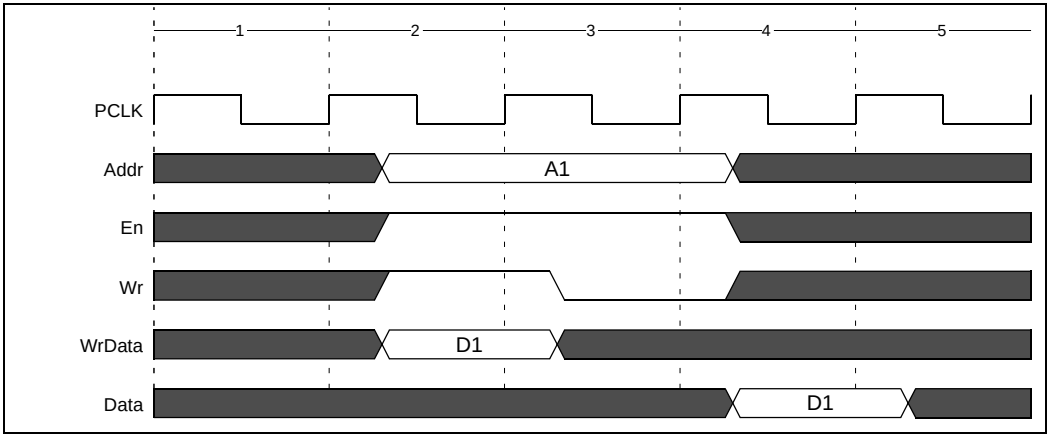


Figure B–218. Write-Read to Memory with 1-Cycle Latency

Figure B–219 makes this behavior more explicit by showing a read of the old data D0 at address A1 just prior to the write of the new data D1 to the same address.

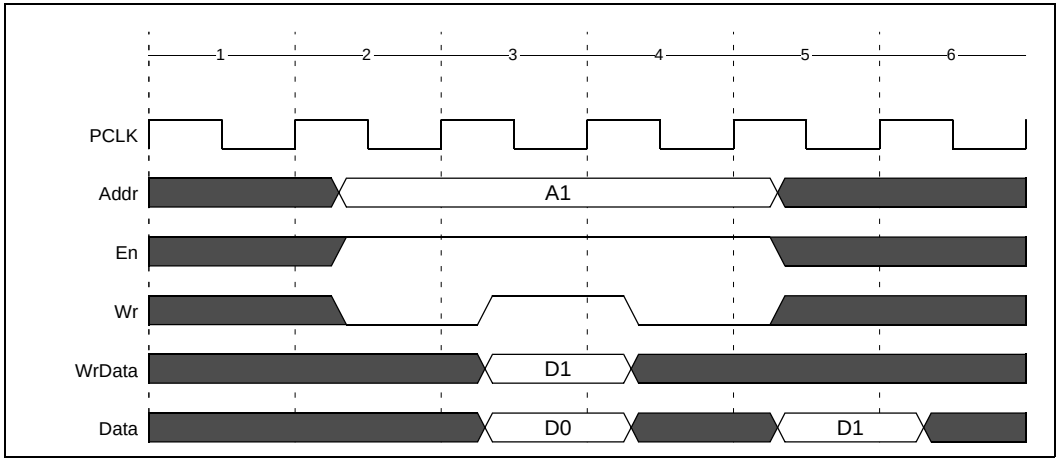


Figure B–219. Read-Write-Read from Memory with 1-Cycle Latency

B.3 Memory with 2-Cycle Latency

Figure B–220 shows a conceptual picture of memory with a 2-cycle latency. A typical non-pipelined synchronous memory module will contain hardware to register its inputs (`Addr`, `En`, `Wr`, WrData), but not its outputs (`Data`). For an Xtensa configuration with a 7-stage pipeline, the memory must be fully pipelined with a register between the data output of each local memory and the data input for the Xtensa processor core's corresponding local-data memory port.

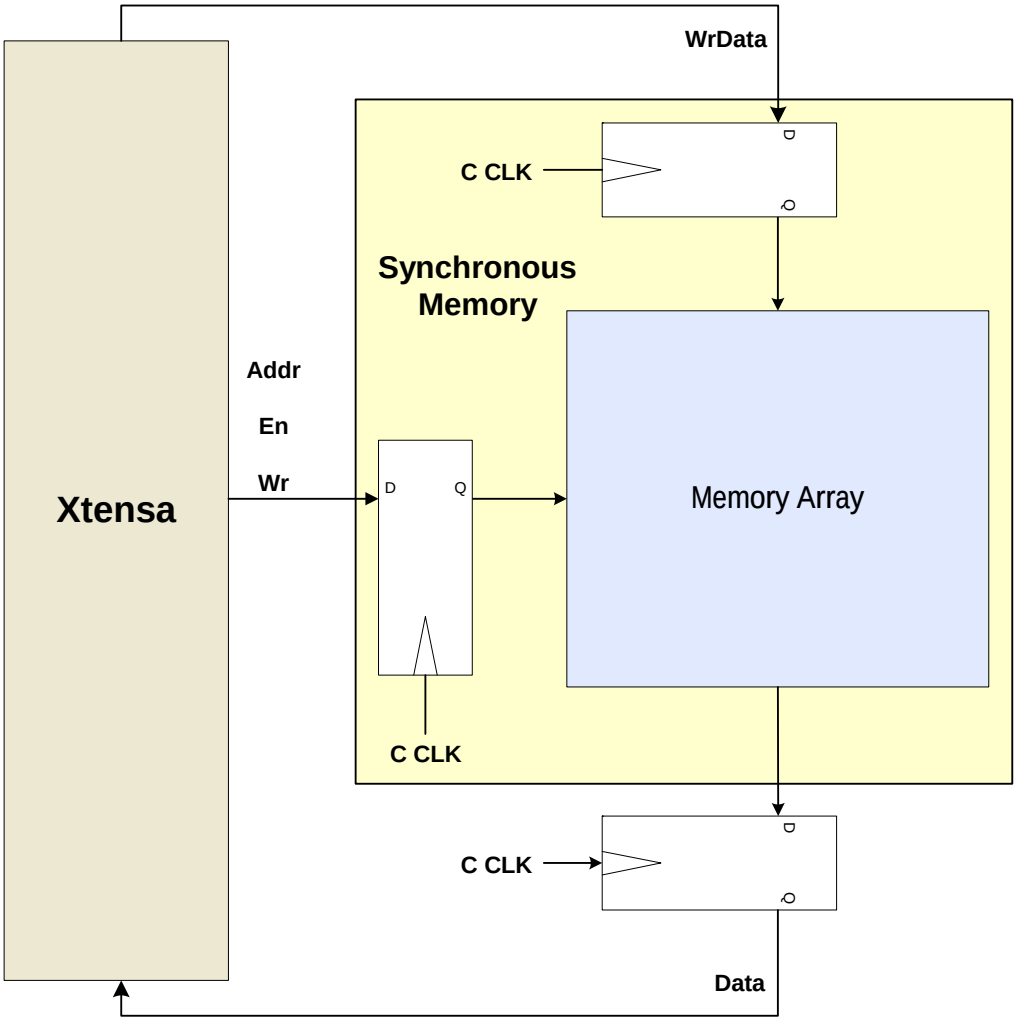


Figure B–220. Single-Port Memory with 2-Cycle Latency

Figure B–221 shows the simplest read for memory with 2-cycle latency. As with memory with 1-cycle latency, the Xtensa processor can either assert or negate memory enable En following the edge where it outputs the read address. If no memory cycles follow, En will be negated. If a read or write follows on the next cycle, En will be asserted.

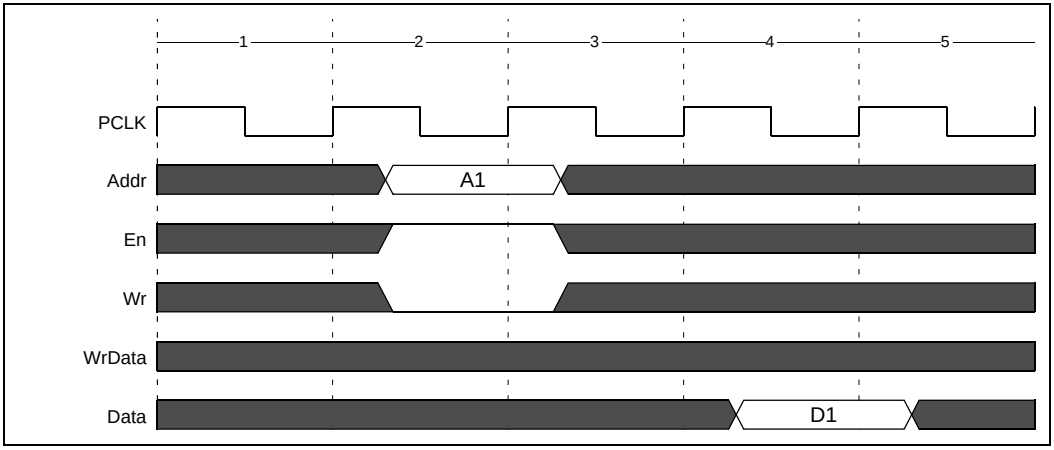


Figure B–221. Read from Memory with 2-Cycle Latency

Figure B–222 shows back-to-back reads from memory with 2-cycle latency. Because the read latency is two cycles, the second read address and enable are asserted one cycle before the processor accepts the first data transfer (D1) from memory.

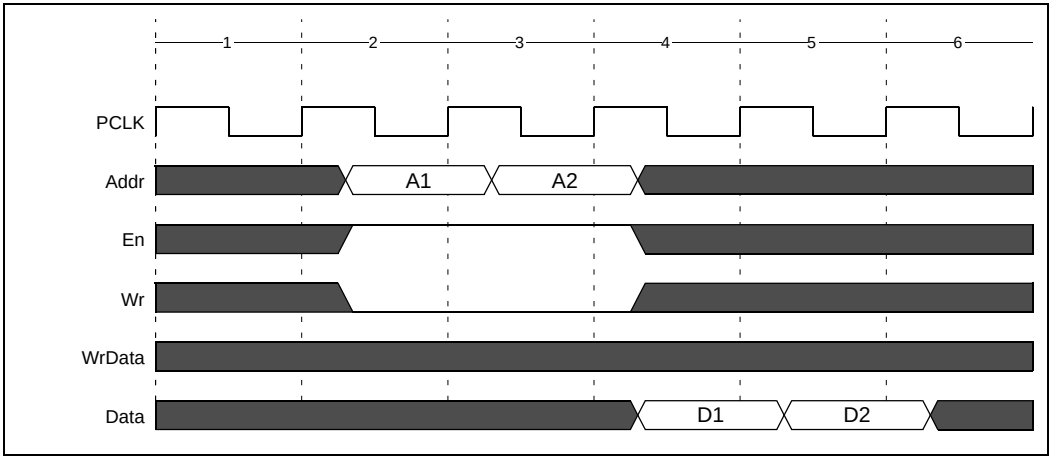


Figure B–222. Back-to-Back Reads from Memory with 2-Cycle Latency

Figure B–223 shows the simplest write for memory with 2-cycle latency.

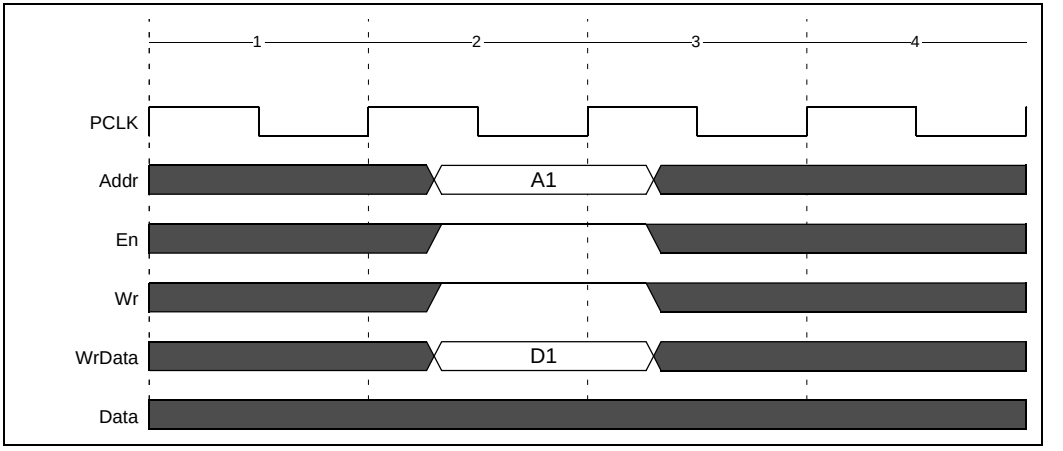


Figure B–223. Write to Memory with 2-Cycle Latency

As with the 1-cycle-latency memory example above, the precise moment of the real update to the memory cell depends on the memory design, but the Xtensa processor expects that a read of the same address in the cycle immediately following the write cycle will yield the new data.

Note: The write operation is required to be the same, whether the memory has 1- or 2-cycle latency.

Figure B–224 shows a write cycle immediately followed by a read cycle for memory with 2-cycle latency.

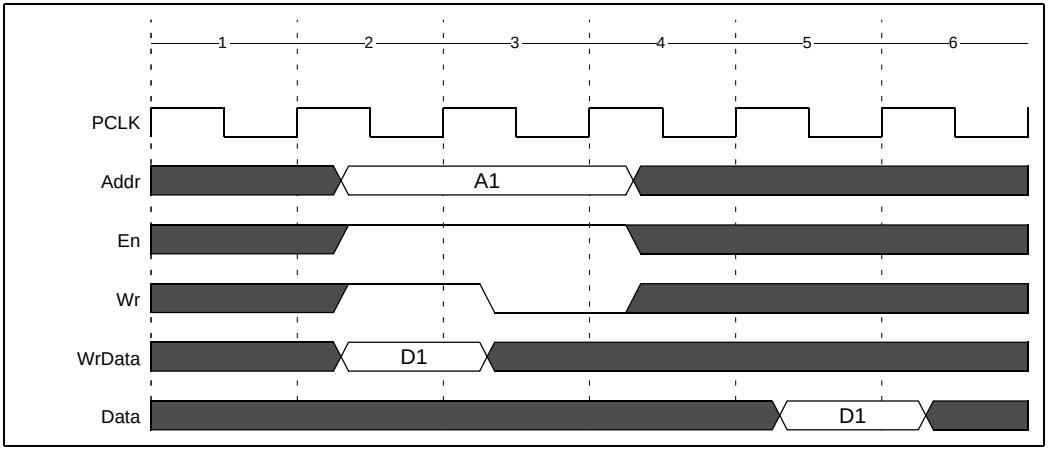


Figure B–224. Write-Read for Memory with 2-Cycle Latency

Figure B–225 shows a read of the old data D0 at address A1 just prior to the write of the new data D1 to the same address. The Xtensa processor outputs the new data D1 before it reads in the old data D0.

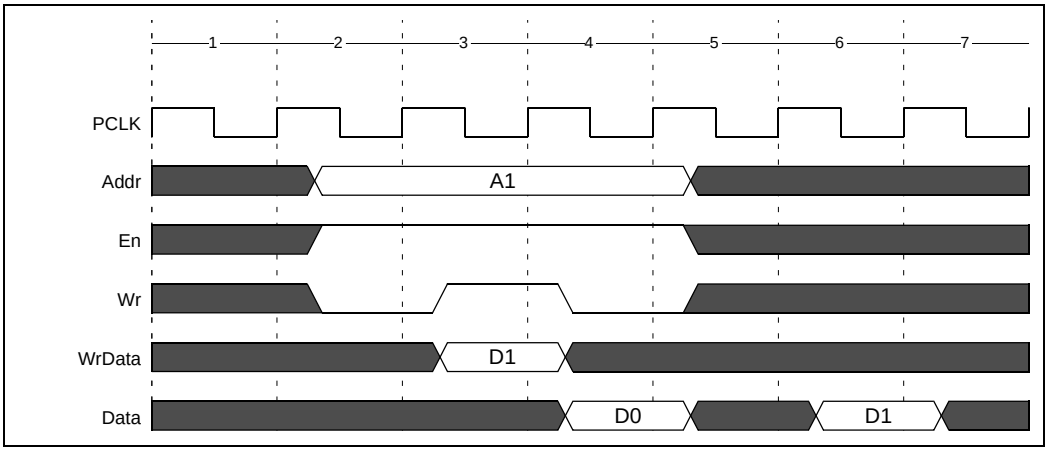


Figure B–225. Read-Write-Read from Memory with 2-Cycle Latency

B.4 Multiported Memory

Local memory may need to connect to multiple Xtensa interface ports. For processor configurations with one load/store unit, there is only one interface to each local memory. However, for processor configurations with two load/store units, local memory (data RAM or data ROM) must be connected to both load/store units through their individual interface ports using a multiplexer. The Xtensa connection-box option can be optionally selected to implement the memory multiplexer for the local-data memories. Alternatively, the local-data memory can be multi-ported with separate ports for each load/store unit. The operational rules described in the above sections apply uniformly, symmetrically, and independently to both memory ports for processor configurations with two load/store units.

- Uniformity means that both interface ports expect the same memory latency—whether the memory has 1- or 2-cycle latency. In other words, based on the behavior of the signals alone, it is not possible to distinguish between the ports.
- Symmetry means that in any of the situations illustrated above—or indeed any set of sequential memory operations—it is possible to attribute a given read or write transaction to either interface port without affecting the outcome of the transaction. For example, a write followed by read to a memory with 2-cycle latency could happen as shown in Figure B–226. In this example, the read from port 1 will get the new data written through port 0. Stable processor operations requires adherence to this rule.

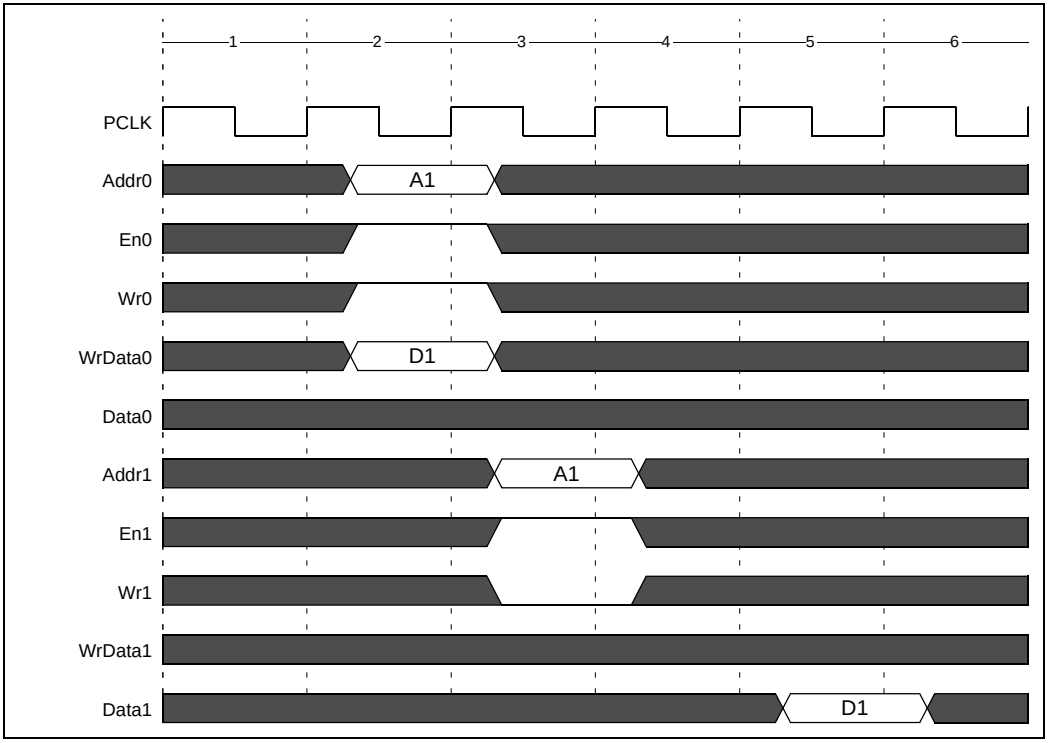


Figure B–226. Write-Read from Different Ports of Dual-Ported Two-Cycle Memory

- Independence means that the operations occurring on one port have no bearing or effect on the operations on the other port—with one exception. The Xtensa processor will never perform writes from different interface ports to the same address during the same cycle.

A read on one port and write on the other port to the same address during the same cycle can occur. In this case, the Xtensa LX7 processor expects to see the new write data value bypassed to the read transaction. Thus, in the case of simultaneous reads and writes, any memory-arbitration circuitry must give priority to the write and return the write data to all simultaneous reads of the same address in order to maintain the required data consistency.

If both simultaneous operations are read transactions, the memory must provide the same data to both interface ports.

If the memory is unable to deal with either of these cases, it can assert the Busy signal to one of the interface ports. More details on the Busy signal are given in the sections describing the data-RAM, data-ROM, and XLMI port interfaces.

C. External Registers

This appendix lists available external registers. These are registers immediately outside the processor core, and accessible through the External Register Interface (ERI). RER and WER instructions executed by the processor pipeline cause respectively, read and write transactions on the ERI.

C.1 External Register Address Space

The external register space has 16 "device" spaces defined in it as shown in Table C–170. Only these spaces are used; other addresses are reserved. Each of the 16 device spaces includes a User portion and a Supervisor portion.

Table C–170. ERI Address Spaces

Beginning Address	Ending Address	Use of Space
0x00000000	0x000FFFFF	Reserved
0x00100000	0x0010FFFF	Device 0 space, Supervisor portion; Debug is device 0
0x00110000	0x0011FFFF	Device 1 space, Supervisor portion; iDMA is device 1
0x001*n*0000	0x001*n*FFFF	Reserved for Device *n* space, Supervisor portion, where *n* is 2 to 16
0x00200000	0x0008FFFF	Reserved
0x00900000	0x0090FFFF	Device 0 space, User portion; Debug has no user registers, therefore this space is unused
0x00910000	0x0091FFFF	Device 1 space, User portion; iDMA is device 1
0x009*n*0000	0x009*n*FFFF	Reserved for Device *n* space, User portion, where *n* is 2 to 16
0x00A00000	0xFFFFFFFF	Reserved

Not all devices have any accessible registers in the User portion; this includes the Debug registers.

For more details on WER/RER privilege and ERI space protection, refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

C.2 List of External Registers

Table C–171 lists all external registers, with sub-heading rows indicating the associated subsystem (*TRAX*, *Performance Monitor*, *OCD, Miscellaneous, Fault, CoreSight* or *iDMA*) and related description. Addresses not shown in this table are reserved and read as zero.

Table C–171. External Registers

Register	ERI Addr	APB Addr	Nexs Reg Addr	Access	Reset Value	Description
TRAX Registers						
TRAXID	0x00100000	0x0000	0x00	RO	*config-specific constant*	ID register (the same as `OCDID`)
TRAXCTRL	0x00100004	0x0004	0x01	R/W	0x00001080	Control register
TRAXSTAT	0x00100008	0x0008	0x02	RO	0x0000*mm*00	Status register (*mm* is log of TraceRAM size)
TRAXDATA	0x0010000C	0x000C	0x03	R*/W	0x0	Data register
TRAXADDR	0x00100010	0x0010	0x04	R/W	0x0	Address register
TRIGGERPC	0x00100014	0x0014	0x05	R/W	0x0	Stop PC
PCMATCHCTRL	0x00100018	0x0018	0x06	R/W	0x0	Stop PC Range
DELAYCNT	0x0010001C	0x001C	0x07	R/W	0x0	Post Stop Trigger Capture Size
MEMADDRSTART	0x00100020	0x0020	0x08	R/W	0x0	Trace Memory Start Address
MEMADDREND	0x00100024	0x0024	0x09	R/W	TraceRAM_words - 1	Trace Memory End Address
Performance Monitor Registers						
PMG	0x00101000	0x1000	0x20	R/W	0x0	Performance counters control register
INTPC	0x00101010	0x1010	0x24	RO	0x0	PC at cycle of last event that caused interrupt
PM0 - PM7	0x00101080-0x0010109C	0x1080 - 0x109C	0x28 - 0x2F	R/W	0x0	Performance counter values
PMCTRL0 - PMCTRL7	0x00101100-0x0010111C	0x1100 - 0x111C	0x30 - 0x37	R/W	0x0	Performance counter control registers
PMSTAT0 - PMSTAT7	0x00101180-0x0010119C	0x1180 - 0x119C	0x38 - 0x3F	R/clr	0x0	Performance counter status registers
OCD Registers						
OCDID	0x00102000	0x2000	0x40	RO	*config-specific constant*	ID Register (the same as `TRAXID`)
DCRCLR	0x00102008	0x2008	0x42	R/clr	0x00600000	Debug Control Register (write to clear)

Table C–171. External Registers (continued)

Register	ERI Addr	APB Addr	Nexs Reg Addr	Access	Reset Value	Description
DCRSET	0x0010200C	0x200C	0x43	R/set	0x00600000	Debug Control Register (write to set)
DSR	0x00102010	0x2010	0x44	R/clr	0x80000000	Debug Status Register
DDR	0x00102014	0x2014	0x45	R/W	0x0	Debug Data Register; for host to/from target transfers
DDREXEC	0x00102018	0x2018	0x46	R*/W	0x0	Alias to `DDR`; executes `DIR` when accessed
DIR0EXEC	0x0010201C	0x201C	0x47	R/W	0x0	Alias to DIR0; executes the instruction when written
DIR0	0x00102020	0x2020	0x48	R/W	0x0	Debug Instruction Register (first 32 bits)
DIR1 - DIR7	0x00102024-0x0010203C	0x2024 - 0x203C	0x49 - 0x4F	R/W	0x0	Debug Instruction Register (other bits)
Miscellaneous Registers						
PWRCTL	0x00103020	0x3020	0x58	R/W	0x0	Power and reset control (resides in Access Port)
PWRSTAT	0x00103024	0x3024	0x59	R/W	interface-specific	Power and reset status (resides in Access Port)
ERISTAT	0x00103028	0x3028	0x5A	R/W	0x0	ERI transaction status
Fault Registers						
FAULTINFO	0x00103030	n/a	n/a	R/W	0x0	See Section 29.3.1 for details. All bits readable, but only 19:16 writable.
AXIECCEN	0x00103034	n/a	n/a	R/W	0x0	▪ Writing 1 enables AXI Parity/ECC checking ▪ Writing 0 disables AXI Parity/ECC checking. ▪ The register affects all AXI ports. When AXI Parity/ECC checking is disabled, parity and ECC code checking on the inputs does not happen, however Xtensa continues to generate parity and ECC codes for its outputs. ▪ Note: ECC enable should only be toggled when there are no AXI transactions in progress
AXIECCSTAT	0x00103038	n/a	n/a	RO	0x0	▪ Bit 0 is set to indicate that an uncorrectable error has occurred ▪ Bit 1 is set to indicatethat a correctable error has occurred
CoreSight Registers						

Table C–171. External Registers (continued)

Register	ERI Addr	APB Addr	Nexs Reg Addr	Access	Reset Value	Description
ITCTRL	0x00103F00	0x3F00	0x60	R/W	0x0	Integration Mode Control Register
CLAIMSET	0x00103FA0	0x3FA0	0x68	R/set	0x0000ffff	Claim Tag Set Register
CLAIMCLR	0x00103FA4	0x3FA4	0x69	R/clr	0x0	Claim Tag Clear Register
LOCKACCESS	0x00103FB0	0x3FB0	0x6C	WO	n/a	Writing 0xC5ACCE55 unlocks internal master access
LOCKSTATUS	0x00103FB4	0x3FB4	0x6D	RO	0x0	Current locking status
AUTHSTATUS	0x00103FB8	0x3FB8	0x6E	RO	0x0	Authentication Status
DEVID	0x00103FC8	0x3FC8	0x72	RO	endianess-based constant	Device ID, including endianness
DEVTYPE	0x00103FCC	0x3FCC	0x73	RO	0x00000015	Device type
Peripheral ID4	0x00103FD0	0x3FD0	0x74	RO	0x00000024	Peripheral ID, including component page size (16 KB)
Peripheral ID5	0x00103FD4	0x3FD4	0x75	RO	0x0	Reserved
Peripheral ID6	0x00103FD8	0x3FD8	0x76	RO	0x0	Reserved
Peripheral ID7	0x00103FDC	0x3FDC	0x77	RO	0x0	Reserved
Peripheral ID0	0x00103FE0	0x3FE0	0x78	RO	0x00000003	Peripheral ID, including part number
Peripheral ID1	0x00103FE4	0x3FE4	0x79	RO	0x00000021	Peripheral ID, including part number and JEDEC code
Peripheral ID2	0x00103FE8	0x3FE8	0x7A	RO	0x0000000f	Peripheral ID, including revision and JEDEC code
Peripheral ID3	0x00103FEC	0x3FEC	0x7B	RO		Peripheral ID, including mask revision
Component ID0	0x00103FF0	0x3FF0	0x7C	RO	0x0000000d	Preamble
Component ID1	0x00103FF4	0x3FF4	0x7D	RO	0x00000090	Component ID, incl. component class (CoreSight)
Component ID2	0x00103FF8	0x3FF8	0x7E	RO	0x00000005	Preamble
Component ID3	0x00103FFC	0x3FFC	0x7F	RO	0x000000b1	Preamble
iDMA Registers						
Settings	0x00*p*10000	n/a	n/a	R/W	0x000000cc	iDMA setting register
Timeout	0x00*p*10004	n/a	n/a	R/W	0x0	Timeout threshold
DescStartAdrs	0x00*p*10008	n/a	n/a	R/W	0x0	Descriptor start address
NumDesc	0x00*p*1000c	n/a	n/a	RO	0x0	The number of descriptors to process
NumDescIncr	0x00*p*10010	n/a	n/a	WO	0x0	Increment to the number of descriptors
Control	0x00*p*10014	n/a	n/a	R/W	0x0	iDMA control register

Table C–171. External Registers (continued)

Register	ERI Addr	APB Addr	Nexs Reg Addr	Access	Reset Value	Description
Privilege	0x00*p*10018	n/a	n/a	R/W	0x0	iDMA privilege register. Writable only in supervisor portion; Readable only is user portion if ERACCESS=1
Status	0x00*p*10040	n/a	n/a	RO	0x0	iDMA status register
DescCurrAdrs	0x00*p*10044	n/a	n/a	RO	0x0	The current descriptor address
DescCurrType	0x00*p*10048	n/a	n/a	RO	0x0	The current descriptor type
SrcAdrs	0x00*p*1004c	n/a	n/a	RO	0x0	Source address from which iDMA is reading from
DestAdrs	0x00*p*10050	n/a	n/a	RO	0x0	Destination address to which iDMA is writing to

Notes with regard to Table C–171 above:

- "R*" means that the read has a side-effect of modifying state associated with the register
- "***p***" is 0x1 for supervisor access and 0x9 for user access. In other words, the three significant nibbles of the ERI address for iDMA registers is 0x001 for supervisor access and 0x009 for user access.

Index

C

Z