SuperH

SuperH^{TM} (SH) 64-Bit RISC Series

SH-5 CPU Core, Volume 1: Architecture

Last updated 22 February 2002

SuperH

Contents

Preface

This document is part of the SuperH SH-5 CPU core documentation suite detailed below. Comments on this or other books in the documentation suite should be made by contacting your local sales office or distributor.

SuperH SH-5 document identification and control

Each book in the documentation suite carries a unique identifier in the form:

05-CC-nnnnn Vx.x

Where, *n* is the document number and *x.x* is the revision.

Whenever making comments on a SuperH SH-5 document the complete identification 05-CC-1000n Vx.x should be quoted.

SuperH SH-5 CPU core documentation suite

The SuperH SH-5 CPU core documentation suite comprises the following volumes:

- SH-5 CPU Core, Volume 1: Architecture (05-CC-10001)
- SH-5 CPU Core, Volume 2: SHmedia (05-CC-10002)
- SH-5 CPU Core, Volume 3: SHcompact (05-CC-10003)
- SH-5 CPU Core, Volume 4: Implementation (05-CC-10004)

1 Overview

1.1 Introduction

The CPU architecture is specified in 4 volumes:

- SH-5 CPU Core, Volume 1: Architecture
- SH-5 CPU Core, Volume 2: SHmedia
- SH-5 CPU Core, Volume 3: SHcompact
- SH-5 CPU Core, Volume 4: Implementation

The first 3 volumes specify the *generic* CPU architecture. This includes the instruction set architecture (ISA) and the mechanisms used for CPU control and configuration. These volumes describe the properties of the architecture which are independent of implementation.

The first implementation of the architecture is called SH-5. Properties specific to this implementation are described separately in the fourth volume.

The CPU architecture does not include descriptions of system features such as the system bus, physical memory system, on-chip peripherals and external interfaces. Also, it does not include descriptions of debug features such as watch-points, tracing and monitoring. These are described in a separate set of documents entitled *SH-5 System Architecture*.

1.2 Instruction set architecture

The architecture provides two instruction sets, called SHmedia and SHcompact, with mechanisms to switch between them.

The SHmedia instruction set represents instructions using a fixed-length 32-bit encoding. SHmedia is used where optimal performance is required, or to access CPU system control and configuration mechanisms.

The SHcompact instruction set represents instructions using a fixed-length 16-bit encoding. SHcompact provides user-mode instruction-level compatibility with previous implementations of the SuperH architecture (see *Section 1.4*). SHcompact is used where code density or compatibility is required.

1.2.1 SHmedia

SHmedia uses a 32-bit instruction encoding. This is less compact than the SHcompact encoding, but has the following advantages:

1. It encodes a three-operand instruction set.
2. Each operand can encode more registers.
3. Constant operands, for immediates and displacements, are wider.
4. It supports a larger instruction set.
5. It has more scope for future extensibility.
6. The encoding is simpler, more regular, and has reduced reliance on mode bits.

These advantages contribute to lower dynamic instruction counts and better performance.

The main properties of the SHmedia instruction set are summarized below.

Large regular register sets

- 64 general-purpose registers, each 64-bit wide (no register banking).
- 64 floating-point registers, each 32-bit wide (no register banking).
- 8 target registers used for branching.
- 64 control registers, each 64-bit wide.
- Other state: program counter, floating-point status and control register, current instruction set mode, configuration registers.

Instruction encoding

- 32-bit encoding: simple, regular and extensible.
- Three-operand format.
- Reasonable-size immediate and displacement fields.
- Mode bits are not used to distinguish instruction opcodes.

Integer instructions

- Operands and operations are 64-bits wide.
- Instructions are well matched to optimizing compilers for high-level languages.
- Efficient support for 32-bit applications.

Branch instructions

- Branch architecture allows program-directed instruction prefetching and buffering to reduce branch penalties.
- Architecturally-defined target registers hold the branch target address.
- Compare-folded branch instructions provide powerful branch semantics.
- Architectural support for static branch prediction.

Load and store instructions

- Architecture supports 64-bit addressing.
- Support for 8-bit, 16-bit, 32-bit and 64-bit data.
- Load and store instructions are separate to arithmetic (this is a load/store architecture).
- Regular addressing modes: register plus register, and register plus scaled immediate.
- Separate instructions provided for misaligned access.
- A 64-bit atomic swap instruction is provided for synchronization.
- Instructions are provided to impose ordering on instruction fetching and memory accesses.
- All load and store instructions support bi-endian data formats. Additionally, an instruction is provided for endianness conversion.

Multimedia instructions

- These operate on multiple data items simultaneously, and improve the performance of multimedia applications.
- An extensive set of data parallel arithmetic and data manipulation operations are provided. These include conversions, addition, subtraction, absolute value, sum of absolute differences, shifts, comparisons, multiplies, multiply accumulate, shuffle, conditional move, permute and extract.

Floating-point instructions

- These are encoded without precision, width and banking mode bits.
- IEEE754 support for single-precision and double-precision representations.
- Non-IEEE754 support including fast handling of denormalized numbers, fused multiply accumulate and special-purpose instructions for graphics.

System, control, configuration, debug and cache

- Instructions are provided to access CPU control and configuration registers.
- Control registers control instruction execution and support event handling.
- Configuration registers are used to configure caches and memory management.
- After a reset, an interrupt, an exception or a panic, execution proceeds in SHmedia.
- A trap instruction is provided to support operating systems.
- A break instruction is provided to support the debug architecture.
- Instructions are provided for cache control.

Future extensibility

- 4 encoding bits are reserved across all instructions to allow extension of the current instruction set.
- A substantial portion of the opcode space is available for future extensions.

1.2.2 SHcompact

SHcompact uses a 16-bit instruction encoding. This is a compact encoding and should be used where high code density is preferred. SHcompact is also used where compatibility with previous implementations of the SH architecture is required. SHcompact does not provide access to CPU system control and configuration mechanisms. The performance of SHcompact code is less optimal than SHmedia code. Thus, SHcompact should not be used for performance-critical code.

The main properties of the SHcompact instruction set are summarized below.

Compatibility

- SHcompact provides user-mode instruction-level compatibility with previous implementations of the SH architecture.
- SHcompact does not support access to CPU system control and configuration mechanisms.
- A precise definition of the compatibility level is described in *Section 1.4*.

Compact register sets

- 16 general-purpose registers, each 32-bit wide.
- 2 banks, each of 16 floating-point registers, each 32-bit wide.
- Other state: program counter, procedure link register, global base register, multiply-accumulate registers, status bits, floating-point status and control register, and floating-point communication register.
- The SHcompact architectural state is mapped onto the SHmedia architectural state. This reduces the overall amount of architectural state, and allows efficient calling conventions to be used between SHmedia and SHcompact code.

Compact instruction encoding

- 16-bit encoding, densely populated encoding, two-operand format.
- High code density.

Integer instructions

- Comprehensive support for integer operations on 32-bit data.

Branch instructions

- Conditional branches based on condition code.
- Delayed branch instructions to hide branch penalties.

Load and store instructions

- Load/store architecture.
- Powerful compact addressing modes.
- Support for bi-endian data formats.

Floating-point instructions

- Mode bits used to control precision, width and banking.
- IEEE754 support for single-precision and double-precision representations.
- Non-IEEE754 support including fast handling of denormalized numbers, fused multiply accumulate and special-purpose instructions for graphics.

System, debug and cache instructions

- A trap instruction is provided to support operating systems.
- A break instruction is provided to support the debug architecture.
- Instructions are provided for cache control.

1.2.3 Mode switch

The architecture provides mechanisms for switching between SHmedia and SHcompact. The mechanisms support mixed mode programming. The expectation is that code which is optimized for speed would be compiled for SHmedia, while code which is optimized for space would be compiled for SHcompact.

Branch control flow

Mode switch can be effected while executing an unconditional branch or a return from exception. The lowest bit of the branch target address denotes the target instruction set. If this bit is 0 the target is SHcompact, otherwise it is SHmedia. Additionally, the procedure call mechanism supports mode switch by saving the return address with the lowest bit conditioned to the mode of the caller. This ensures that a return to the caller will use the correct mode.

One possible software arrangement is to support mode switching at the point of procedure call and return. With this convention, the granularity for selecting mode is the procedure. The compiler, or perhaps the user, selects which procedures are compiled for SHmedia and which for SHcompact. The SHcompact registers are architecturally mapped onto the SHmedia registers. This allows efficient calling conventions between the two modes.

Another software arrangement could exploit mode switch within procedures. The granularity for selecting mode would then be the basic block.

The SHcompact instruction set contains extended semantics, relative to previous implementations of the architecture, in order to support mode switching. Compatibility is described in *Section 1.4*.

Event handling

When a reset, an interrupt, an exception or a panic occurs, the CPU arranges for the execution of a software handler. This handler is launched in privileged mode using the SHmedia instruction set, regardless of the mode of the previous context.

SHmedia provides mechanisms for system control and these are used by software during handler entry and exit. After appropriate handling, software often arranges for the previous context to be restarted. The restart sequence also executes in SHmedia, and will switch mode, as required, to restart the previous context.

In between the handler entry and exit sequences, the handler can freely switch between SHmedia and SHcompact. Both SHmedia and SHcompact instructions can be executed in privileged mode. Only SHmedia has access to the full architectural state, and only SHmedia implements system control and configuration instructions.

1.3 CPU control and configuration

The architecture provides powerful mechanisms for control and configuration. These have been designed to support operation using a minimal run-time environment or using a real-time kernel.

1. The memory architecture supports bi-endian data formats. The selection between little endian and big endian organizations is determined by an implementation-specific mechanism at power-on reset. The selected endianness is then consistently used by instructions that access memory.
2. The architecture supports the handling of reset, interrupt, exception and panic conditions.
3. The architecture supports a wide range of memory management implementations.
4. The architecture supports a wide range of cache implementations.

1.4 SH compatibility model

SHcompact provides user-mode instruction-level compatibility with previous implementations of the SuperH instruction set. The intent is that SuperH user-mode programs, that exercise only architecturally-defined properties, can be executed without recompilation using the SHcompact instruction set.

SHcompact provides compatibility with the SH-4 instruction set specifically. The SH-4 instruction set is a superset of the SH-3, SH-2 and SH-1 instruction sets, though there are some subtleties in the degree of compatibility through these evolutions of the SuperH architecture. In cases of ambiguity, SHcompact maintains compatibility with SH-4.

The extent of the compatibility is described in this section.

1.4.1 User-mode compatibility

Every user-mode instruction supported by SH-4 is implemented in the SHcompact instruction set. The implementation has identical user-visible semantics, apart from the following special cases.

Mode switch instructions

SHcompact supports a mode switch mechanism whereas SH-4 does not. The semantics of the following SHcompact instructions have been extended relative to SH-4:

- BRAF, BSRF (PC-relative branching with offset specified using a register).
- JMP, JSR (absolute branching with target specified using a register).
- RTS (return from sub-routine with target specified in the procedure link register).

These are all of the SHcompact instructions that perform unconditional branching where the target is specified in a register. The least significant bit of the target register is used by SHcompact to indicate the target mode, whereas on SH-4 this lowest bit indicates instruction misalignment.

If this bit is 0, then SHcompact and SH-4 have identical behavior. If this bit is 1 and the second least significant bit is 0, then SHcompact causes the target to be executed as an SHmedia instruction whereas SH-4 causes a misaligned instruction exception. If the two least significant bits are both 1, then SHcompact and SH-4 have the same behavior, and both cause a misaligned instruction exception.

This change does not affect compatibility with programs that execute entirely in user-mode. However, it limits compatibility with SH-4 programs that use instruction misalignment exceptions to enter privileged mode.

TAS.B instruction

The semantics of the TAS.B instruction in SHcompact have been reduced relative to SH-4. The SH-4 TAS.B instruction guarantees atomicity with respect to all memory accesses from all memory users.

The SHcompact TAS.B instruction provides a test-and-set operation which is atomic from the CPU perspective. This instruction cannot be interrupted during its operation. However, atomicity is not provided with respect to accesses from other memory users. There is no special treatment for TAS.B regarding the cache, and it behaves in the same way as a memory read followed by a memory write. Depending on the cache behavior, it is possible for the TAS.B accesses to be completed in the cache with no external memory activity.

The SHcompact semantics continue to support the use of TAS.B to synchronize between software threads executing on the same CPU. It cannot be used to synchronize with other memory users or hardware devices. This change is consistent with the provision of instruction-level compatibility; the architecture does not provide compatibility at the system level.

The SHmedia SWAP.Q instruction (see *Section 6.5.1: Atomic swap on page 98*) provides an atomic read-modify-write on external memory, and should be used for synchronization with other memory users.

Floating-point instructions

SHcompact provides a set of special-purpose floating-point instructions which are used to accelerate certain applications where strict IEEE754 conformance is not required. These special-purpose FPU instructions give approximate results. The degree of approximation is defined by an architected error bound that specifies the maximum amount that an implementation result can differ from the infinite-precision result.

Both SHcompact and SH-4 honor the same architected error bound for approximate FPU instructions. However, there is no guarantee that the results in SHcompact and SH-4 are bit-exact relative to each other.

Cache instructions

The MOVCA.L instruction (move with cache block allocation) and the OCBI instruction (operand cache block invalidate) implicitly reveal the cache line size to a program exploiting them. This is because the amount of memory that is affected by these instructions is determined by the cache line size.

For implementations of this architecture that support a 32-byte cache line, the user-visible behavior of these instructions is compatible with SH-4.

1.4.2 Limits of compatibility

This section lists some of the limits of the SH compatibility model.

Privileged instructions

SHcompact does not support any SH-4 instructions that require privileged execution. The privileged SH-4 instructions are:

- LDC and STC instructions (excluding those that access GBR)
- LDTLB
- RTE
- SLEEP

Execution of these instructions in SHcompact causes a reserved instruction exception to be raised regardless of privilege.

Reserved instruction encodings

Execution of the SHcompact instruction encoding 0xFFFD causes a reserved instruction exception to be raised regardless of privilege. This is the same behavior as SH-4.

The behavior of SHcompact instruction encodings which are neither defined nor reserved is implementation specific. The following choices are available to implementations when such an encoding is executed:

- An implementation can cause a reserved instruction exception to be raised.
- An implementation can exhibit some implementation-defined behavior (though it must not provide any behavior that would breach the privilege model).

Properties of the address space

For compatibility with an existing program binary, it is typically necessary to execute that program in an effective address space using memory management features to recreate an appropriate memory environment. This requires some compatibility in memory management capability to allow the appropriate address translations to be created. Many properties of the MMU are implementation-specific, and thus the degree of compatibility achieved here depends on the implementation.

In most cases, effective address calculation is performed using modulo 64-bit integer arithmetic. This contrasts with SH-4 which uses modulo 32-bit integer arithmetic. The behavior of SHcompact for effective address calculations that overflow a 32-bit address is different from SH-4.

The specific cases are:

- SHcompact load and store instructions that involve effective address calculation using registers: the available addressing forms include register plus register, register plus constant, global base register plus constant and register with pre-decrement. These calculations are performed with modulo 64-bit integer arithmetic. An exception is generated where the calculation causes an effective address to be generated outside the implemented effective address space.

  Note that register indirect and register with post-increment are also provided. In both cases the effective address of the access is taken from a source register with no calculation and there is no opportunity for the access to use an effective address outside the implemented effective address space.

  Also note that PC plus constant addressing does not use modulo 64-bit integer arithmetic. In this case the calculation of the effective address is performed using 32-bit integer arithmetic. The resulting effective address is converted to a sign-extended 32-bit range. This gives an address which is always inside the implemented effective address space.

- SHcompact branch instructions that involve effective address calculation: the available addressing forms include PC plus constant and PC plus register. These calculations are performed with modulo 64-bit integer arithmetic. An exception is generated where the calculation causes an effective address to be generated outside the implemented effective address space.

  SHcompact's conditional branch instructions use PC plus constant addressing. For these instructions, the range check on the calculated effective address is performed, and an exception taken where required, even when the branch is not taken.

These cases can be caught by an exception and handled in system software. Software approaches for handling these cases are described in *Section 2.9.3: SHcompact memory on page 29*.

The CPU architecture does not impose any properties on the layout or contents of the physical address map. Thus, the CPU architecture does not specify whether the physical address map, and any memory-mapped mechanisms available within it, are compatible with SH-4. These properties are implementation dependent.

Exceptions

There is no binary compatibility provided for exception handling software. Thus SH-4 exception handlers must be ported and recompiled.

The exception launch mechanism has been designed so that the user-visible behavior of instruction execution, including the effects of exception handling, can be compatible with SH-4. This compatibility requires appropriate system software.

This is achieved as follows. For each exception taken by some SH-4 code, there will be a corresponding exception taken by the corresponding SHcompact code. For most exceptions there is a direct correspondence with those provided on SH-4. However, in some cases the architecture arranges exception launch somewhat differently. The user-visible SH-4 behavior can be reconstructed by system software.

There are two important exclusions to the above model:

- There is no requirement for any consistency between the timing of translation miss exceptions in SHcompact compared with SH-4. This is because translation misses should be handled, by an operating system, transparently with respect to program execution. This gives considerable flexibility in the organization of the memory management unit.
- The mode switch mechanism causes some SH-4 instruction misalignment exceptions to be treated as mode switch directives. The effect of this change in behavior is described in *Section 1.4.1*.

2 Architectural state

2.1 Overview

This chapter describes the architectural state of the CPU. The execution of instructions causes changes to the architectural state. Some state changes can also be caused by external agents, though these mechanisms are beyond the scope of the CPU Architecture Manual.

2.2 User and privileged operation

The execution model distinguishes operation in user mode from operation in privileged mode. Architectural state is divided into user state and privileged state. Similarly, instructions are divided into user instructions and privileged instructions. In user mode, only user instructions and user state are available. In privileged mode, all instructions and all state are available. Thus, in privileged mode the available mechanisms are a superset of those available in user mode.

This model allows systems to be constructed such that the privileged parts of the system can be protected against the user parts of the system. Additionally, systems can be constructed containing multiple independent user parts, such that they are protected against each other.

2.3 Effective addresses

Some items of architectural state hold effective addresses. The number of implemented bits in effective addresses is implementation defined. Further information is given in *Section 3.8: Effective address representation on page 49*.

2.4 Notation

The following notation is used to describe ranges of integral values, where i and j are integers and $i \leq j$:

- [i, j] is the range of integers between i and j, where i and j are included.
- (i, j) is the range of integers between i and j, where i and j are excluded.
- [i, j) is the range of integers between i and j, where i is included and j is excluded.
- (i, j] is the range of integers between i and j, where i is excluded and j is included.

The following notation is used for exponentiation, where i and j are integers and $j \geq 0$:

- i^j denotes the integral result of raising i to the power of j.

2.5 User state

The user state is summarized in *Table 1* and *Table 2*.

User state	Description
MD (implicit state)	User (0) or privileged (1) mode
ISA (implicit state)	Instruction set architecture (0 for SHcompact, 1 for SHmedia)
PC	64-bit program counter
R_i where i is in [0, 63]	64 x 64-bit general-purpose registers
TR_i where i is in [0, 7]	8 x 64-bit target address registers
CR_i where i is in [32, 63]	32 x 64-bit user-accessible control registers
MEM[i] where i is in [0, 2^{64})	Memory

Table 1: User state (general-purpose)

User state is described in more detail in the following sections.

User state	Description
FPSCR	32-bit floating-point status and control register
FR_i where i is in [0, 63]	64 x 32-bit floating-point registers
DR_{2i} where i is in [0, 31]	32 x 64-bit floating-point registers
FP_{2i} where i is in [0, 31]	32 pairs of 32-bit floating-point registers
FV_{4i} where i is in [0, 15]	16 vectors of 4 x 32-bit floating-point registers
$MTRX_{16i}$ where i is in [0, 3]	4 matrices of 16 x 32-bit floating-point registers Note that FR, DR, FP, FV and MTRX provide different views of the same architectural state.

Table 2: User state (floating-point)

2.5.1 Mode: MD

MD distinguishes user mode and privileged mode. The value of MD is considered user state because it implicitly affects the behavior of instructions executed in user mode. User mode instructions can neither read nor write MD. Instructions executing in user mode do not have a direct means of determining that they are running in user mode.

If a privileged instruction is executed in user mode, then a reserved instruction exception is raised. The attempted privilege violation can be dealt with appropriately in privileged mode.

MD is actually a synonym for the MD field in the status register (SR). See *Section 2.7* for details.

2.5.2 Instruction set architecture: ISA

ISA distinguishes whether instructions are decoded and executed using the SHmedia instruction set or using the SHcompact instruction set.

ISA is implicit state, and cannot be read or written directly. The current value of ISA determines how the current instruction is decoded. Any particular instruction must be encoded for a particular ISA. Each instruction is implicitly associated with the ISA that it executes under.

While ISA cannot be written directly, mechanisms are provided to allow ISA to be updated upon an unconditional branch or a return from exception.

2.5.3 Program counter: PC

PC contains the program counter of the currently executing instruction. The PC is 64 bits wide, though the number of implemented higher bits is implementation defined.

2.5.4 General-purpose registers: R

R denotes the set of general-purpose registers. Most instructions use general-purpose registers to supply integer source values and to hold integer destination values.

There are 64 general-purpose registers, each containing 64 bits. All reads from R_{63} return zero, and all writes to R_{63} are discarded. All other general-purpose registers have conventional read/write behavior.

2.5.5 Target registers: TR

TR denotes the set of 8 target address registers. These are used to hold the target program counter value for branches. Each target register is 64 bits wide, though the number of implemented higher bits is implementation defined.

2.5.6 User-accessible control registers: CR

CR denotes the set of control registers. CR_{32} to CR_{63} are user-accessible control registers, and contain CPU control state accessible in either user or privileged mode.

There are 32 user-accessible control registers, each containing 64 bits. An implementation need not provide all of these control registers, and not all bits of the provided control registers need be implemented. Additionally, control register state, by its very nature, does not have simple read/write semantics. The content and behavior of control registers are described individually for each control register.

Mechanisms to access control registers are described in *Chapter 9: SHmedia system instructions on page 163*. The actual control registers are described in *Chapter 15: Control registers on page 207*.

2.5.7 Memory: MEM

MEM denotes memory. Memory is accessed through an effective address space. Instructions are fetched from memory, and data can be accessed in memory using load and store instructions.

The effective address space allows 2^{64} bytes of memory to be addressed. The amount of implemented space is implementation defined, though the number of addressable bytes is always a power of 2, say 2^{neff}, where neff is in the range [32, 64]. Where neff<64 the implementation provides a sign-extended subset of the 64-bit effective address space. Further information is given in *Section 3.8: Effective address representation on page 49*.

2.5.8 Floating-point status and control register: FPSCR

FPSCR contains the floating-point status and control register. This is used principally to control the rounding mode and exception behavior of floating-point arithmetic. The FPSCR is 32 bits wide, though some of these bits are architecturally reserved. FPSCR is described in *Chapter 8: SHmedia floating-point on page 135*.

2.5.9 Floating-point registers: FR, DR, FP, FV, MTRX

FR denotes the set of single-precision floating-point registers. Each register in FR can hold one single-precision floating-point value using a representation consistent with the IEEE754 standard. The format is defined in *Section 3.4.2: Single-precision format on page 34*. The FR set has 64 registers, each containing 32 bits.

DR denotes the set of double-precision floating-point registers. Each register in DR can hold one double-precision floating-point value using a representation consistent with the IEEE754 standard. The format is defined in *Section 3.4.3: Double-precision format on page 35*. DR provides a different view of the same architectural state provided by FR. The DR set has 32 registers, each containing 64 bits. The mapping from the DR set onto the FR set is achieved as follows:

- The high half of DR_{2i} where i is in [0, 31] maps onto FR_{2i}.
- The low half of DR_{2i} where i is in [0, 31] maps onto FR_{2i+1}.

Additionally, the FR set can be accessed as pairs, as vectors or as 16-element matrices of single-precision floating-point values (see *Figure 1*):

- There are 32 x 2-element pairs called FP_{2i} where i is in [0, 31].
  Each pair consists of registers FR_{2i} and FR_{2i+1}.

- There are 16 x 4-element vectors called FV_{4i} where i is in [0, 15]. Each vector consists of registers $FR_{4i,}$ $FR_{4i+1,}$ FR_{4i+2} and FR_{4i+3}.
- There are 4 x 16-element matrices called $MTRX_i$ where i is in [0, 3]. Each matrix consists of registers FR_{16i}, $FR_{16i+1,}$ FR_{16i+2} through to FR_{16i+15}.

The numbering of the elements within a pair, vector and matrix is shown in *Figure 2*. The 16-element matrix can also be viewed as a 4x4 matrix with column-major format as shown in *Figure 3*.

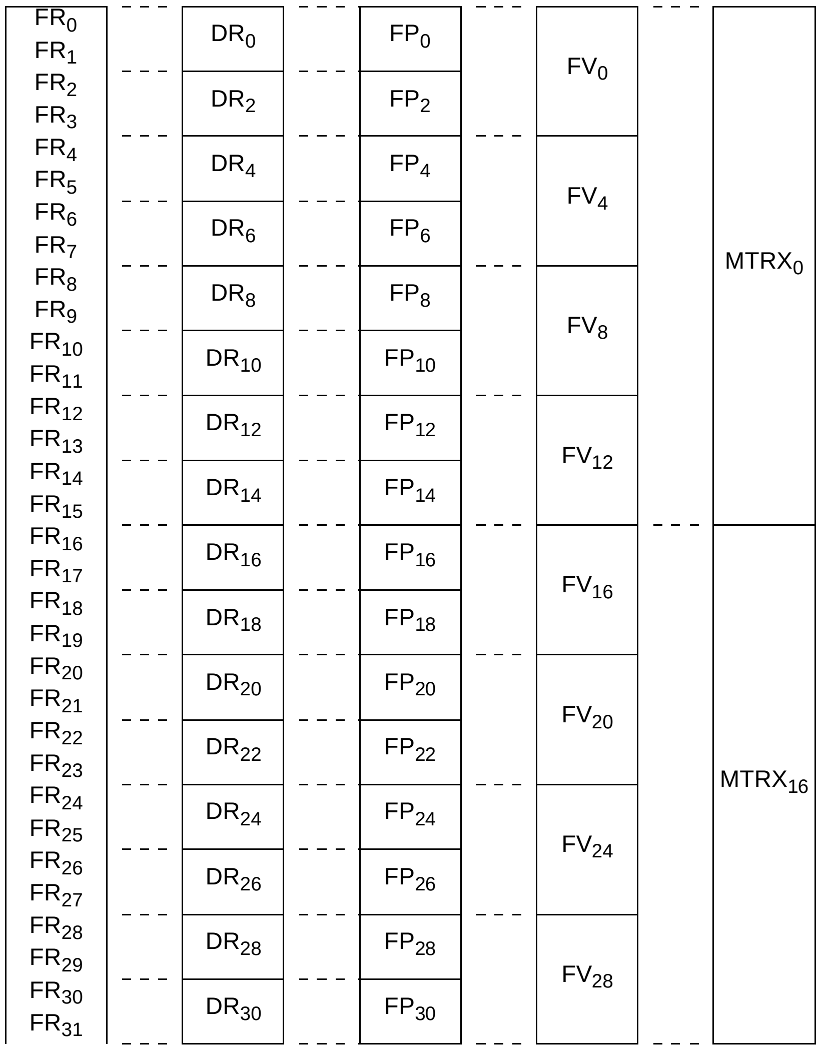


Figure 1: Alternate views of the floating-point register set

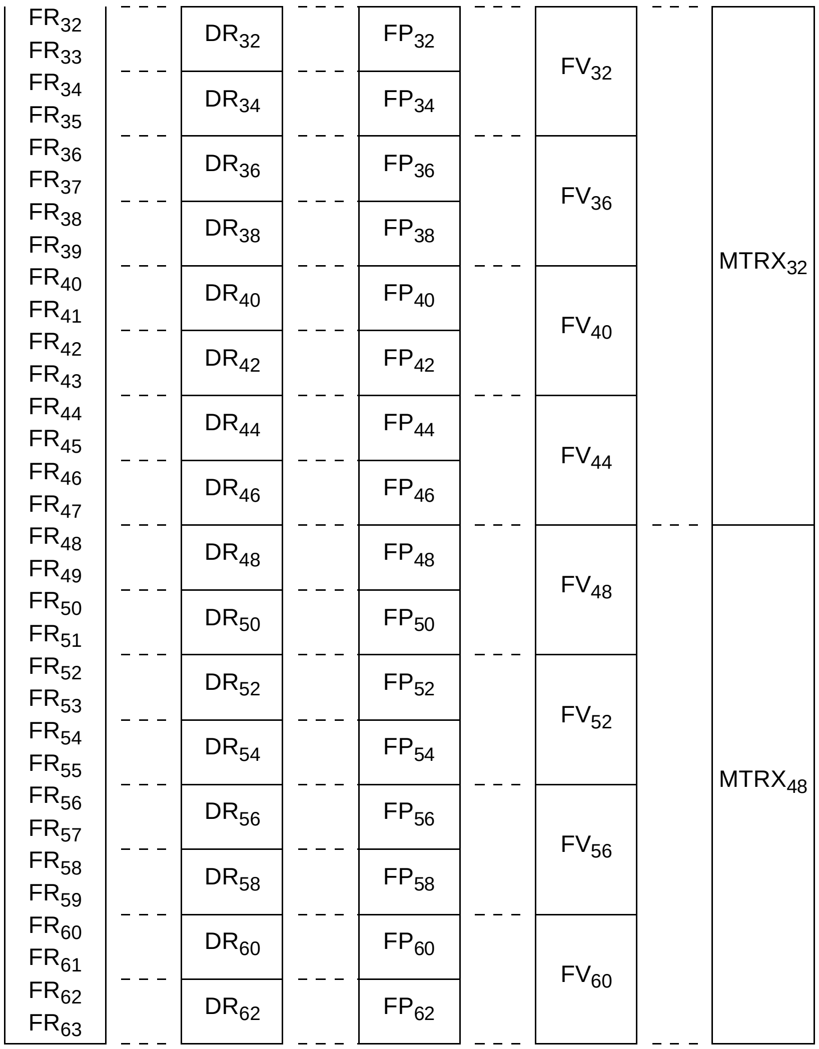

FR32 … FR63
DR32 DR34 DR36 DR38 DR40 DR42 DR44 DR46 DR48 DR50 DR52 DR54 DR56 DR58 DR60 DR62
FP32 FP34 FP36 FP38 FP40 FP42 FP44 FP46 FP48 FP50 FP52 FP54 FP56 FP58 FP60 FP62
FV32 FV36 FV40 FV44 FV48 FV52 FV56 FV60
MTRX32
MTRX48

Figure 1: Alternate views of the floating-point register set

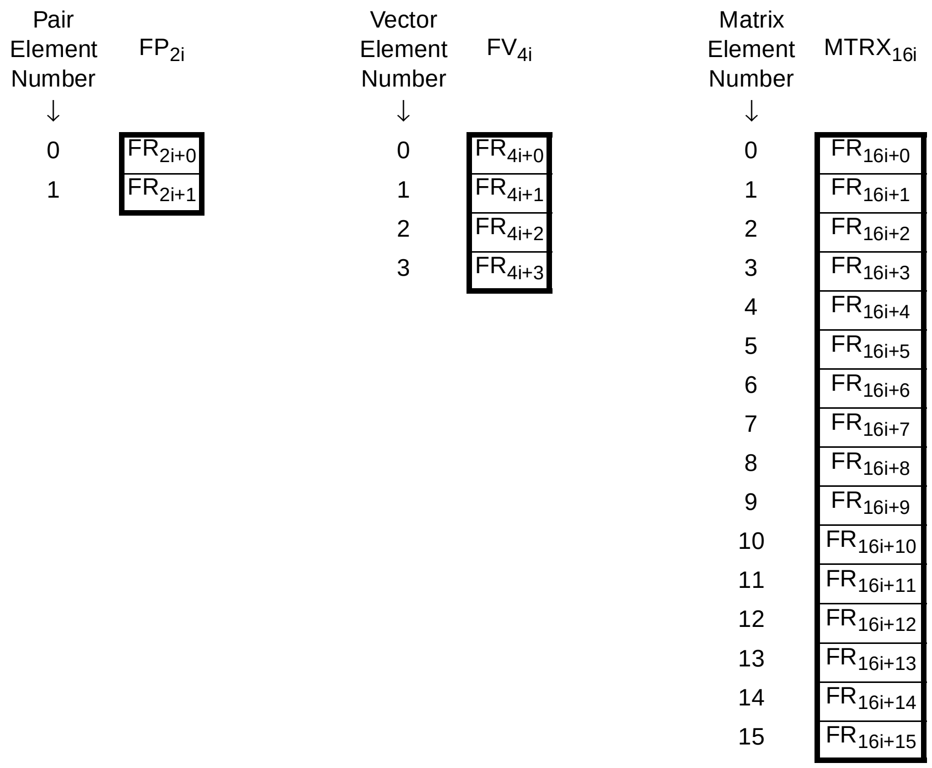


Figure 2: FP, FV and MTRX element numbering

Column Number → / Row Number ↓	0	1	2	3
0	FR_{16i+0}	FR_{16i+4}	FR_{16i+8}	FR_{16i+12}
1	FR_{16i+1}	FR_{16i+5}	FR_{16i+9}	FR_{16i+13}
2	FR_{16i+2}	FR_{16i+6}	FR_{16i+10}	FR_{16i+14}
3	FR_{16i+3}	FR_{16i+7}	FR_{16i+11}	FR_{16i+15}

Figure 3: MTRX 4x4 element numbering

2.6 Privileged state

The privileged state is summarized in *Table 3*.

Privileged State	Description
CR_i where i is in [0, 31]	32 x 64-bit privileged control registers
CFG[i] where i is in [0, 2^{32})	Configuration register space

Table 3: Privileged state

Privileged state is described in more detail in the following sections. The key differences between control registers and configuration registers are that control registers hold scalar state and are accessed through a control register set, while configuration registers typically hold arrays of state and are accessed through a configuration register space.

2.6.1 Privileged control registers: CR

CR denotes the set of control registers. CR_0 to CR_{31} are privileged control registers, and contain CPU control state accessible only in privileged mode. The trap, exception, interrupt and reset mechanisms use this state.

There are 32 privileged control registers, each containing 64 bits. An implementation need not provide all of these control registers, and not all bits of the provided control registers need be implemented. Additionally, control register state, by its very nature, does not have simple read/write semantics. The content and behavior of control registers are described individually for each control register.

Mechanisms to access control registers are described in *Chapter 9: SHmedia system instructions on page 163*. The actual control registers are described in *Chapter 15: Control registers on page 207*.

The status register (SR) is a privileged control register that implicitly affects the execution of instructions by the current thread of execution. SR is introduced in *Section 2.7*.

2.6.2 Configuration registers: CFG

CFG denotes configuration register space. Configuration register space is completely independent of memory, and is accessed using different instructions. Configuration space allows 2^{32} configuration registers to be addressed, though the amount of implemented space is implementation defined.

Configuration registers are 64 bits wide, though not all bits of the provided configuration registers need be implemented. Additionally, configuration register state, by its very nature, does not have simple read/write semantics. The content and behavior of configuration registers are described individually for each configuration register.

Configuration registers are used for direct access to implementation specific state, such as cache and MMU state. Configuration registers are highly implementation dependent and are therefore not defined in this document. Mechanisms to access configuration registers are described in *Chapter 9: SHmedia system instructions on page 163*.

2.7 The status register

The status register (SR) is a control register that contains fields to control the behavior of instructions executed by the current thread of execution. The layout of SR is shown in *Figure 4*.

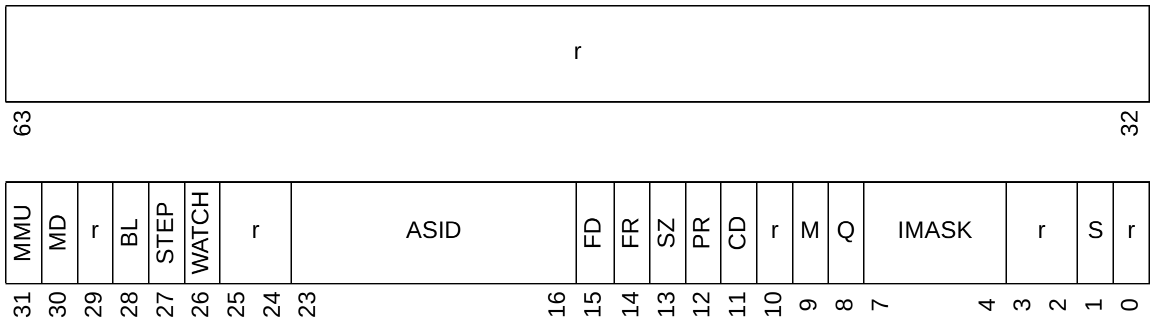


Figure 4: SR (upper 32 bits and lower 32 bits shown separately)

The specification of SR is given in *Section 15.2.1: SR on page 210*. In summary, the fields of SR are used as follows:

- S, Q and M are used during the execution of SHcompact instructions. They are described in *Chapter 11: SHcompact integer instructions on page 171* and in *Volume 3 Chapter 2: SHcompact instruction set*.
- FR, SZ and PR are used to provide additional opcode qualification of SHcompact floating-point instructions. They are described in *Chapter 13: SHcompact floating-point on page 189*.
- IMASK contains 4 bits to allow the CPU to be set to one of 16 priority levels for masking interrupts. It is described in *Chapter 16: Event handling on page 221*.
- CD controls whether a read of the clock tick counter from user mode returns the value of the clock tick counter (when clear) or zero (when set). It is described in *Section 15.2.10: CTC on page 218*.
- FD controls whether the floating-point instructions are enabled (when clear) or disabled (when set). It is described in *Chapter 8: SHmedia floating-point on page 135* and *Chapter 13: SHcompact floating-point on page 189*.
- ASID indicates the address space identifier of the current thread and is used by the memory management architecture. It is described in *Chapter 17: Memory management on page 271*.

- WATCH controls whether watchpoints are disabled (when clear) or enabled (when set). It is described in *Chapter 16: Event handling on page 221*.
- STEP controls whether single-stepping is disabled (when clear) or enabled (when set). It is described in *Chapter 16: Event handling on page 221*.
- BL controls whether exceptions, traps and interrupts are allowed (when clear) or blocked (when set). It is described in *Chapter 16: Event handling on page 221*.
- MD controls whether instructions are executed in user mode (when clear) or in privileged mode (when set). It implicitly affects instruction execution, and is described in *Section 2.5.1: Mode: MD on page 15*.
- MMU controls whether the MMU is disabled (when clear) or enabled (when set). It is described in *Chapter 17: Memory management on page 271*.
- The ‘r’ field indicates reserved bits.

In privileged mode, the value of SR can be read and written using the SHmedia GETCON and PUTCON instructions (see *Section 9.3.2*). However, PUTCON cannot be used to modify the ASID, WATCH, STEP, MD and MMU fields. These can be modified using the RTE instruction (see *Section 9.2* and *Section 16.7*).

2.8 Register subsets

The general-purpose and floating-point register sets are each divided into 8 subsets, and each of theses subsets consists of 8 consecutive registers. For i in the range [0,7]:

- General-purpose register subset i, $GPRS_i$, contains R_{8i} to R_{8i+7} inclusive.
- Floating-point register subset i, $FPRS_i$, contains FR_{8i} to FR_{8i+7} inclusive.

Each subset is associated with a dirty bit. All dirty bits are held in the user-accessible status register (USR), which is described in *Section 15.2.11: USR on page 219*. Dirty bits can be read and written by software, either in user mode or in privileged mode.

When an instruction is executed that writes to a modifiable general-purpose register or floating-point register, then the dirty bit for the subset containing that register will be set to 1. However, note that a write to R_{63} is not required to set its dirty bit since the value of R_{63} is always 0. The hardware can set dirty bits under other circumstances, but it never automatically clears the dirty bits. This is only achieved by explicit software action.

This mechanism allows an operating system to keep track of which register subsets have been written to, and this information can be used to optimize context switches. For example, an operating system can be arranged such that if a subset of registers has not been modified since the last context switch, then that subset of registers does not get saved out to memory on the next context switch. This is particularly important for threads that execute entirely in SHcompact since these can only use a small proportion of the available register state.

Additionally, both user and privileged mode software can clear the dirty bits. This could be used, for example, to indicate that the values in a subset of registers are no longer required, and that the operating system need not consider that subset as dirty.

2.9 SHcompact state

SHcompact provides only user instructions. All SHcompact instructions can be executed in user mode or in privileged mode. State accessed by SHcompact instructions is user state and is architecturally mapped onto the user architectural state already described. SHcompact uses a separate naming convention for its state.

2.9.1 SHcompact non-floating-point register state

The mapping of the non-floating-point SHcompact state onto the architectural state is shown in *Table 4*. The status register, SR, is not itself directly accessible in SHcompact, though the S, M and Q flags are accessible through their own names.

Names of SHcompact state	Description of state	Architectural state name
PC	Program counter	Lower 32 bits of PC
R_i where i is in [0, 15]	General-purpose registers	Lower 32 bits of R_i
GBR	Global base register	Lower 32 bits of R_{16}
MACL	Multiply-accumulate low	Lower 32 bits of R_{17}
MACH	Multiply-accumulate high	Upper 32 bits of R_{17}
PR	Procedure link register	Lower 32 bits of R_{18}

Table 4: Mapping of SHcompact non-floating-point state

Names of SHcompact state	Description of state	Architectural state name
T	Condition code flag	Bit 0 of R_{19}
S	Multiply-accumulate saturation flag	SR.S
M	Divide-step M flag	SR.M
Q	Divide-step Q flag	SR.Q
(no names - not visible)	These registers are used as scratch state during the execution of SHcompact instructions. When any SHcompact instruction is executed, the value of these registers becomes architecturally undefined (even if the instruction causes an exception).	R_{20} to R_{23} inclusive TR_0 to TR_3 inclusive

Table 4: Mapping of SHcompact non-floating-point state

Special care must be taken by SHmedia software to ensure that unused upper bits have appropriate values when switching into SHcompact. The requirements are defined in *Volume 3 Chapter 1: SHcompact specification*.

2.9.2 SHcompact floating-point register state

The mapping of the SHcompact floating-point state is shown in *Table 5* and *Table 6*.

Names of SHcompact state	Description of state		Architectural state name
FPSCR.RM	Floating-point status	Rounding mode	FPSCR.RM
FPSCR.FLAG		Exception flags	FPSCR.FLAG
FPSCR.ENABLE		Exception enables	FPSCR.ENABLE
FPSCR.CAUSE		Exception causes	FPSCR.CAUSE
FPSCR.DN		Denormalization mode	FPSCR.DN
FPSCR.PR		Precision of operation	SR.PR
FPSCR.SZ		Size of data transfer	SR.SZ
FPSCR.FR		Bank selection	SR.FR

Table 5: Mapping of non-banked SHcompact floating-point state

Names of SHcompact state	Description of state	Architectural state name
FPUL	FPU communication register	FR_{32}
(no name - not visible)	This register is used as scratch state during the execution of SHcompact floating-point instructions. When any SHcompact floating-point instruction is executed, the value of this register becomes architecturally undefined (even if the instruction causes an exception).	FR_{33}

Table 5: Mapping of non-banked SHcompact floating-point state

The definition of FPSCR differs between SHmedia and SHcompact for the PR, SZ and FR flags. For SHmedia, these flags are contained within SR and do not appear in FPSCR. For SHcompact, these flags appear in FPSCR only and SR is not directly accessible. Coherency of these flags is maintained automatically and no special software action is required.

SHcompact provides two banks of floating-point registers. The status register contains a flag called SR.FR which determines which bank is viewed using the regular floating-point register names and which as the extended bank. The setting of this flag determines how the banked SHcompact floating-point state maps onto the SHmedia floating-point state.

Names of SHcompact state	Description of state	Architectural state name (when SR.FR = 0)	Architectural state name (when SR.FR = 1)
FR_i where i is in [0, 15]	Single-precision registers	FR_i	FR_{i+16}
DR_{2i} where i is in [0, 7]	Double-precision registers	DR_{2i}	DR_{2i+16}
	Single-precision register pairs	FP_{2i}	FP_{2i+16}
FV_{4i} where i is in [0, 3]	Single-precision vector	FV_{4i}	FV_{4i+16}

Table 6: Mapping of banked SHcompact floating-point state

Names of SHcompact state	Description of state	Architectural state name (when SR.FR = 0)	Architectural state name (when SR.FR = 1)
XF_i where i is in [0, 15]	Single-precision extended registers	FR_{i+16}	FR_i
XD_{2i} where i is in [0, 7]	Double-precision extended registers	DR_{2i+16}	DR_{2i}
	Single-precision extended register pairs	FP_{2i+16}	FP_{2i}
XMTRX	Single-precision extended register matrix	$MTRX_{16}$	$MTRX_0$

Table 6: Mapping of banked SHcompact floating-point state

2.9.3 SHcompact memory

SHcompact provides access to a 32-bit effective address space. This is mapped onto the 64-bit effective address space provided by the architecture using a sign-extended convention. The mapping is shown in *Table 7*.

SHcompact Effective Address Range	Architectural Effective Address Range
[0x00000000, 0x7FFFFFFF]	[0x0000000000000000, 0x000000007FFFFFFF]
[0x80000000, 0xFFFFFFFF]	[0xFFFFFFFF80000000, 0xFFFFFFFFFFFFFFFF]

Table 7: Mapping of SHcompact effective address space

Although the amount of implemented effective address space is implementation defined, every implementation provides at least a 32-bit effective address space in a sign-extended subset of the 64-bit space (see *Section 2.5.7: Memory: MEM on page 17*). This ensures that the full SHcompact 32-bit effective address space is available on all implementations.

In most cases, the effective address calculation in SHcompact load, store and branch instructions is actually performed using modulo 64-bit integer arithmetic. It is possible for these SHcompact instructions to access some effective addresses in the range [0x0000000080000000, 0xFFFFFFFF7FFFFFFF] through effective address calculation near the discontinuity in the effective address space. Such accesses are considered as programming errors and should be avoided in SHcompact programs. The specific cases are listed in *Section 1.4.2: Limits of compatibility on page 10*.

It is possible for system software to catch exceptions for these accesses. If the implementation provides exactly 32 bits of effective address space, then accesses outside of the SHcompact effective address space will cause an address error. If the implementation provides more than 32 bits of effective address space, then the address translation mechanism can be used to force translation misses for addresses outside of the SHcompact effective address space. It is possible to fix up these exceptions in system software so that addressing is fully compatible with SH-4.

Alternatively, in the case of an implementation providing more than 32 bits of effective address space, the translation mechanism could be used to ensure that these addresses are silently mapped onto the appropriate part of the SHcompact effective address space. This also gives full addressing compatibility with SH-4.

Further information on effective addresses and pointers can be found in *Section 3.8: Effective address representation on page 49* and *Section 3.10: Pointer representation on page 51*. Data address exceptions are described in *Section 16.11.3: Data address exceptions on page 244*, and translation mechanisms are described in *Chapter 17: Memory management on page 271*.

3 Data representation

3.1 Introduction

This chapter describes the representation of data in the general-purpose registers, the floating-point registers and in memory. The same data representations apply in SHmedia and SHcompact to support efficient parameter passing between these modes.

The SHmedia instruction set is much richer than the SHcompact instruction set. As a consequence, operations on some data representations, particularly 64-bit wide data, can be inefficient when performed using the SHcompact instruction set.

This chapter also describes the effective address space and pointer representation. Finally, the chapter describes the conventions used for control and configuration registers.

3.2 Bit conventions

A register is a collection of binary bits, where each bit can take a value of 0 or 1. The number of bits in a register, say b, varies according to the type of register. The bits in a register are numbered from 0 (the least significant bit) up to b-1 (the most significant bit). Registers are depicted with the most significant bit left-most on the page, and the least significant bit right-most.

A bitfield is a contiguous subset of bits taken from a register. If a register contains b bits, then a bitfield of that register is specified as the set of bits starting at bit number s and ending at bit number e where the range from s to e is inclusive. The following relationship holds between s, e and b: $0 \leq s \leq e < b$. The bits in the bitfield are numbered from 0 (the least significant bit) up to e-s (the most significant bit). Bit-fields are depicted with the same ordering convention as registers.

3.3 Data types

Data types are provided to support modern high-level programming languages such as ANSI C, C++ and Java, and standards such as IEEE754 floating-point arithmetic. The directly supported types are summarized in *Table 8*.

Data type	SHmedia support	SHcompact support
Unsigned 8-bit integer	Load/store	Load/store
Signed 8-bit integer	Load/store	Load/store
Unsigned 16-bit integer	Load/store	Load/store
Signed 16-bit integer	Load/store	Load/store
Unsigned 32-bit integer	Load/store/arithmetic	Load/store/arithmetic
Signed 32-bit integer	Load/store/arithmetic	Load/store/arithmetic
32-bit pointer	Load/store/arithmetic/addressing	Load/store/arithmetic/addressing
32-bit floating-point	Load/store/IEEE754-arithmetic	Load/store/IEEE754-arithmetic
Unsigned 64-bit integer	Load/store/arithmetic	No direct support
Signed 64-bit integer	Load/store/arithmetic	No direct support
64-bit pointer	Load/store/arithmetic/addressing	No direct support
8 x 8-bit multimedia data	Load/store/multimedia	No direct support
4 x 16-bit multimedia data	Load/store/multimedia	No direct support
2 x 32-bit multimedia data	Load/store/multimedia	No direct support
64-bit floating-point	Load/store/IEEE754-arithmetic	Load/store/IEEE754-arithmetic

Table 8: Supported data types

Signed data is always held using a two’s complement representation. SHcompact requires multiple instructions to load 8-bit or 16-bit data into an unsigned integer format.

The architecture contains general-purpose registers and floating-point registers. Each data type can be held in general-purpose registers or floating-point registers, though the available operations will vary considerably.

3.4 IEEE754 floating-point numbers

3.4.1 Values

An IEEE754 floating-point number contains three fields: a sign (s), an exponent (e) and a fraction (f) in the following format:

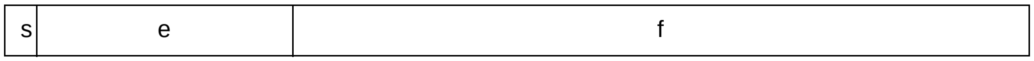


Figure 5: IEEE754 floating-point representations

The sign, s, is the sign of the represented number. If s is 0, the number is positive. If s is 1, the number is negative.

The exponent, e, is held as a biased value. The relationship between the biased exponent, e, and the unbiased exponent, E, is given by e = E+bias, where bias is a fixed positive number. The unbiased exponent, E, takes any value in the range $[E_{min}-1, E_{max}+1]$. The minimum and maximum values in that range, $E_{min}-1$ and $E_{max}+1$, designate special values such as positive zero, negative zero, positive infinity, negative infinity, denormalized numbers and “Not a Number” (NaN).

The fraction, f, specifies the binary digits that lie to the right of the binary point. A normalized floating-point number has a leading bit of 1 which lies to the left of the binary point. A denormalized floating-point number has a leading bit of 0 which lies to the left of the binary point. The leading bit is implicitly represented; it is determined by whether the number is normalized or denormalized, and is not explicitly encoded. The implicit leading bit and the explicit fraction bits together form the significand of the floating-point number.

The value, v, of a floating-point number is determined as follows:

NaN: if $E = E_{max} + 1$ and $f \neq 0$, then v is Not a Number irrespective of the sign s

Positive or negative infinity: if $E = E_{max} + 1$ and $f = 0$, then $v = (-1)^s (\infty)$

Normalized number: if $E_{min} \leq E \leq E_{max}$, then $v = (-1)^s\ 2^E(1.f)$

Denormalized number: if $E = E_{min} - 1$ and $f \neq 0$, then $v = (-1)^s\ 2^{Emin}(0.f)$

Positive or negative zero: if $E = E_{min} - 1$ and $f = 0$, then $v = (-1)^s\ 0$

The architecture supports two IEEE754 basic floating-point number formats: single-precision and double-precision.

3.4.2 Single-precision format

A single-precision floating-point value has 32 bits:

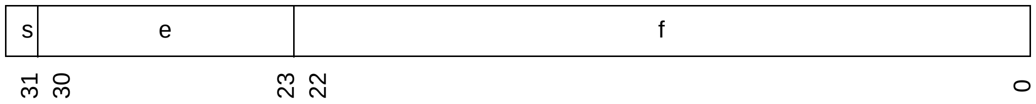


Figure 6: Single-precision floating-point representation

The single-precision format parameters are:

Single-precision format parameter	Value
Width in bits	32
Exponent width in bits	8
Significand bits (fraction bits plus an implicit leading bit)	24
Exponent bias	+127
E_{max}	+127
E_{min}	-126

Table 9: Single-precision floating-point parameters

The types of single-precision floating-point values and their representation are:

Single-precision value type	Representation
+INF (positive infinity)	0x7F800000
+NORM (positive normalized number)	0x00800000 to 0x7F7FFFFF
+DENORM (positive denormalized number)	0x00000001 to 0x007FFFFF
+0.0 (positive zero)	0x00000000
-0.0 (negative zero)	0x80000000
-DENORM (negative denormalized number)	0x807FFFFF to 0x80000001
-NORM (negative normalized number)	0xFF7FFFFF to 0x80800000

Table 10: Single-precision floating-point values

Single-precision value type	Representation
-INF (negative infinity)	0xFF800000
sNaN (signalling not-a-number)	0x7FC00000 to 0x7FFFFFFF, and 0xFFFFFFFF to 0xFFC00000
qNaN (quiet not-a-number)	0x7F800001 to 0x7FBFFFFF, and 0xFFBFFFFF to 0xFF800001

Table 10: Single-precision floating-point values

A NaN, in the single-precision format, is represented as:

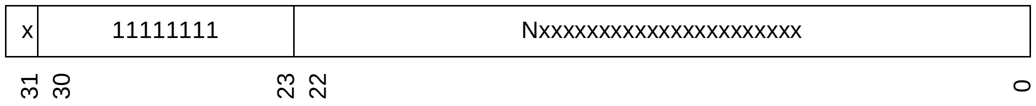


Figure 7: Single-precision NaN values

A single-precision floating-point number is a NaN if the exponent field contains the maximum representable value and the fraction is non-zero, regardless of the value of the sign. In the figure above, x can have a value of 0 or 1. If the most significant bit of the fraction (N, in the figure above) is 1, the value is a signaling NaN (sNaN) otherwise the value is a quiet NaN (qNaN).

3.4.3 Double-precision format

A double-precision value has 64 bits:

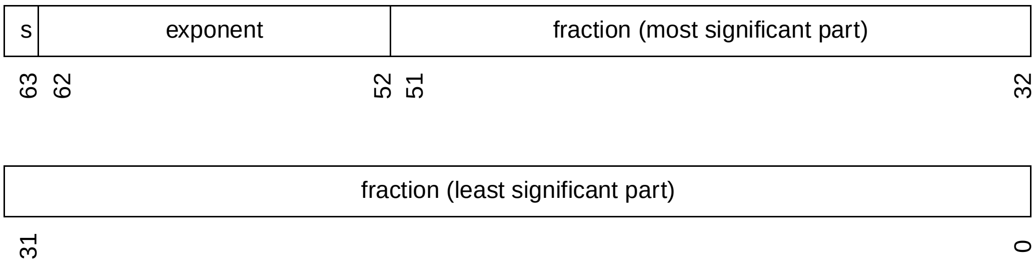

s
exponent
fraction (most significant part)
63
62
52
51
32
fraction (least significant part)
31
0

Figure 8: Double-precision floating-point representation

The double-precision format parameters are:

Double-precision format parameter	Value
Width in bits	64
Exponent width in bits	11
Significand bits (fraction bits plus an implicit leading bit)	53
Exponent bias	+1023
E_{max}	+1023
E_{min}	-1022

Table 11: Double-precision floating-point parameters

The types of double-precision floating-point values and their representation are:

Double-precision value type	Representation
+INF (positive infinity)	0x7FF00000_00000000
+NORM (positive normalized number)	0x00100000_00000000 to 0x7FEFFFFF_FFFFFFFF
+DENORM (positive denormalized number)	0x00000000_00000001 to 0x000FFFFF_FFFFFFFF
+0.0 (positive zero)	0x00000000_00000000
-0.0 (negative zero)	0x80000000_00000000
-DENORM (negative denormalized number)	0x800FFFFF_FFFFFFFF to 0x80000000_00000001
-NORM (negative normalized number)	0xFFEFFFFF_FFFFFFFF to 0x80100000_00000000
-INF (negative infinity)	0xFFF00000_00000000
sNaN (signalling not-a-number)	0x7FF80000_00000000 to 0x7FFFFFFF_FFFFFFFF, and 0xFFFFFFFF_FFFFFFFF to 0xFFF80000_00000000
qNaN (quiet not-a-number)	0x7FF00000_00000001 to 0x7FF7FFFF_FFFFFFFF, and 0xFFF7FFFF_FFFFFFFF to 0xFFF00000_00000000

Table 12: Double-precision floating-point values

A NaN, in the double-precision format, is represented as:

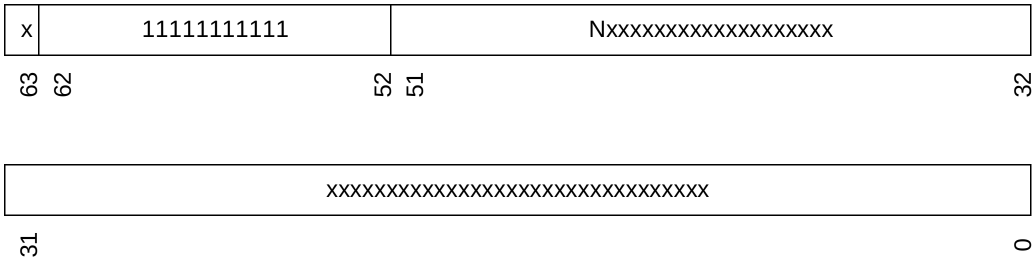


Figure 9: Double-precision NaN values

A double-precision floating-point number is a NaN if the exponent field contains the maximum representable value and the fraction is non-zero, regardless of the value of the sign. In the figure above, x can have a value of 0 or 1. If the most significant bit of the fraction (N, in the figure above) is 1, the value is a signaling NaN (sNaN) otherwise the value is a quiet NaN (qNaN).

3.5 Data formats for general-purpose registers

General-purpose registers contain 64 bits. These are numbered from 0 (least significant bit) to 63 (most significant bit).

Each supported data type consists of a whole number of bytes, where a byte is 8 contiguous bits. When a data type of n bytes is held in a general-purpose register, it occupies bits in the inclusive range from bit number 0 up to bit number 8n-1.

The order of byte numbering within the data type depends on the endianness. For little endian, bytes within an n-byte data type are numbered starting from 0 for the least significant byte up to n-1 for the most significant byte. For big endian, bytes within the data type are numbered starting from 0 for the most significant byte of the data type up to n-1 for the least significant byte. The byte numbering indicates the order in which the bytes are held in memory; the byte number always increases with the byte address.

The representation obeys the following conventions:

- Data types are always packed into general-purpose registers such that they start at the least significant bit. Any bits unused by a data type occur at the more significant end.
- 8-bit and 16-bit unsigned integer values have their upper unused bits set to 0.

- 8-bit and 16-bit signed integer values have their upper unused bits set to sign extensions of bit 7 and 15, respectively.
- 32-bit unsigned integer, 32-bit signed integer and 32-bit floating-point values have their upper unused bits set to sign extensions of bit 31. Instructions are provided to operate on these data types when held in this sign-extended format. This is known as a sign-extended 32-bit representation. The representation of a 32-bit object with the upper unused bits set to 0 (a zero-extended 32-bit representation) is not favored by the architecture.
- 64-bit data types have no unused bits and are held in general-purpose registers in the obvious way.

A multimedia type consists of e elements each containing b bits. Multimedia types occupy all 64 bits of the general-purpose register, so e multiplied by b is always 64. The elements are always numbered in the direction of increasing significance regardless of endianness. Element i, where i is in [0, e), occupies bits in the range [ib, ib+b).

The 32-bit floating-point type uses the IEEE754 single-precision format. The 64-bit floating-point type uses the IEEE754 double-precision format. The bit layout of the floating-point types is described in *Chapter 8: SHmedia floating-point on page 135*.

The data representation of the supported data types is summarized in the following diagrams. The shaded cells indicate the data bits carried by each format; unused bits are zero or sign extensions as defined by the conventions described previously.

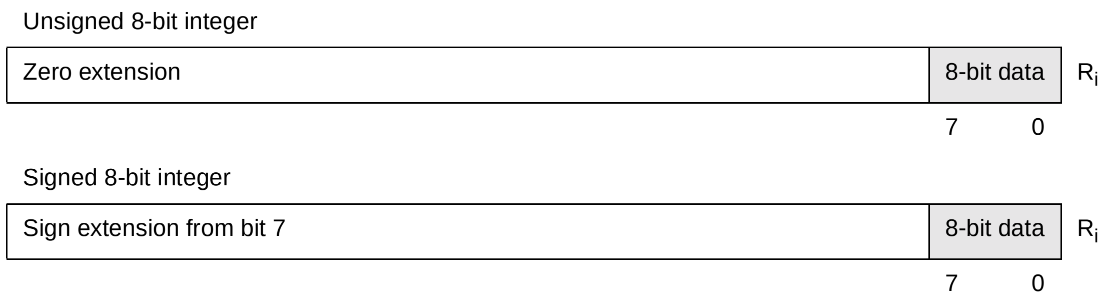
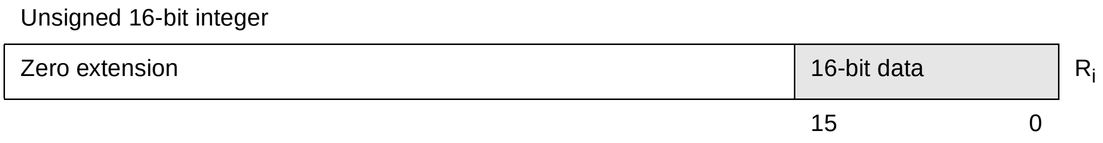


Figure 10: Data representation in general-purpose registers (8-bit types)



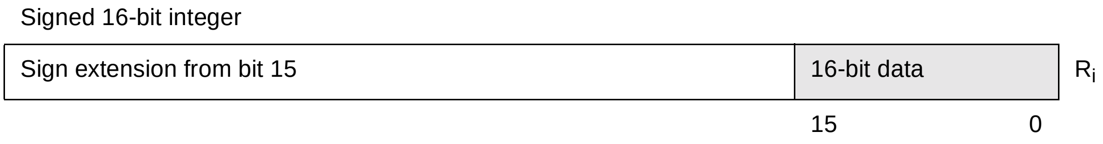


Figure 11: Data representation in general-purpose registers (16-bit types)

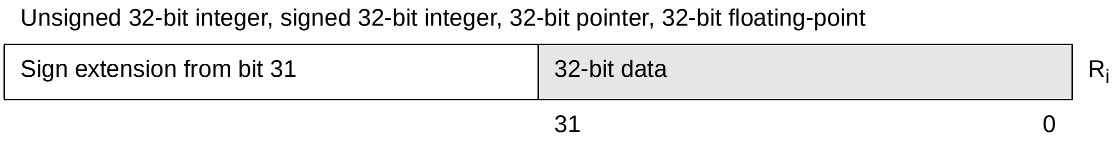

Unsigned 32-bit integer, signed 32-bit integer, 32-bit pointer, 32-bit floating-point
Sign extension from bit 31
32-bit data
R_i
31
0

Figure 12: Data representation in general-purpose registers (32-bit types)

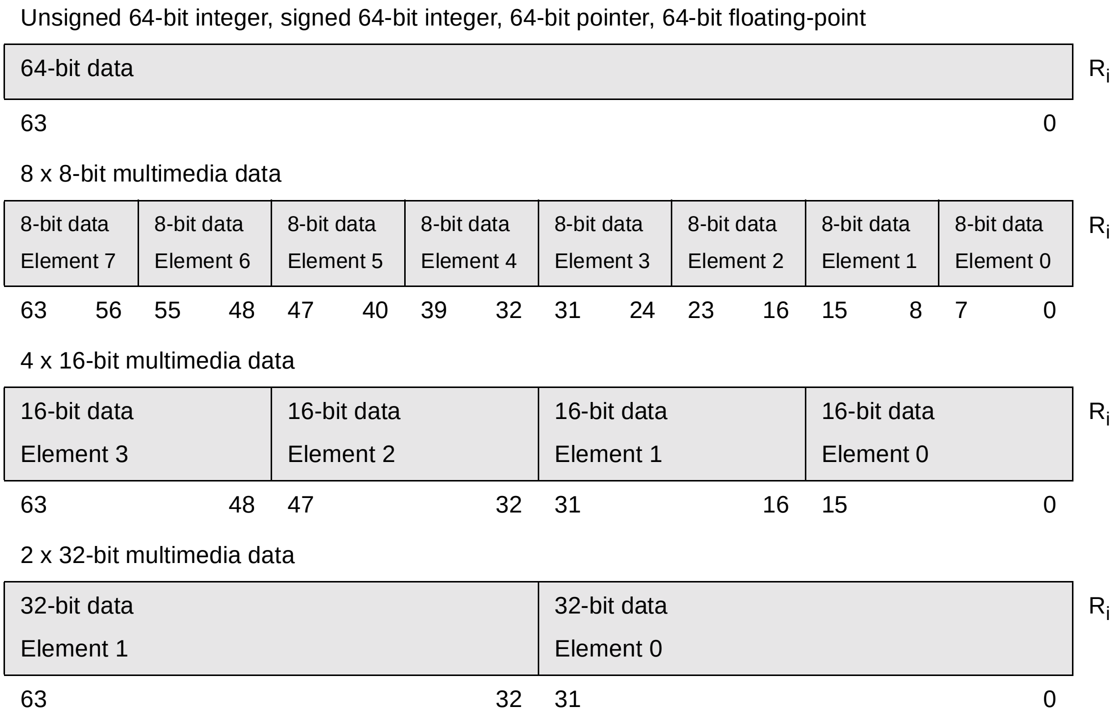


Figure 13: Data representation in general-purpose registers (64-bit types)

Each instruction in SHmedia or SHcompact is designed to work with operands of a particular data representation. Care must be taken with upper unused bits to ensure that they have the correct values since many instructions rely on this property. In particular, most SHcompact instructions are designed for 32-bit data types. These instructions require that their source operands are in a sign-extended 32-bit representation for correct behavior. These instructions then ensure that their destination operands are in a sign-extended 32-bit representation.

Where both SHmedia and SHcompact support a particular data type, the data representation is the same. It is possible to arrange appropriate software conventions that eliminate overhead when passing register-held values between programs executing in different modes.

SHcompact programs are not aware of the presence or value of the upper 32 bits of general-purpose registers. SHmedia programs, however, must take care to condition the upper 32 bits appropriately. This is particularly important when interfacing with SHcompact code.

3.6 Data formats for floating-point registers

The floating-point registers can be viewed as a set of 64 x 32-bit single-precision registers, or as a set of 32 x 64-bit double-precision registers.

The bits of a single-precision register are numbered from 0 (least significant bit) to 31 (most significant bit). For little endian, the bytes are numbered from 0 (least significant byte) to 3 (most significant byte). For big endian, the bytes are number from 0 (most significant byte) to 3 (least significant byte).

The bits of a double-precision register are numbered from 0 (least significant bit) to 63 (most significant bit). For little endian, the bytes are numbered from 0 (least significant byte) to 7 (most significant byte). For big endian, the bytes are number from 0 (most significant byte) to 7 (least significant byte).

Each double-precision register, DR_{2i} where i is in [0, 31], uses the same architectural state as a pair of single-precision registers, FR_{2i} and FR_{2i+1}. The upper 32 bits of DR_{2i} are held in FR_{2i}, and the lower 32 bits of DR_{2i} are held in FR_{2i+1}. The ordering of this split is independent of endianness. Endianness does not affect the register set organization; it determines the byte numbering and the memory representation only.

The mapping of data types into these registers is as follows:

- A single-precision register, FR_i where i is in [0, 63], can hold any supported 32-bit data type. The single-precision representation is defined in terms of the equivalent representation in a general-purpose register. The single-precision register representation simply holds the same bit-pattern as the lower 32 bits of the general-purpose register representation (the upper 32 bits are not represented).
- A double-precision register, DR_{2i} where i is in [0, 31], can hold any supported data type containing 32 or 64 bits. The double-precision representation is defined in terms of the equivalent representation in a general-purpose register. The double-precision register representation simply holds the same bit-pattern as the general-purpose register representation (all 64 bits are represented).
- A pair of single-precision registers, FP_{2i} where i is in [0, 31], can hold two instances of any supported 32-bit data type. A pair consists of two registers: FR_{2i} contains element 0 and FR_{2i+1} contains element 1 of the pair. Each element in the pair uses the same data representations as those used by single-precision registers.

Floating-point registers can also hold 8-bit and 16-bit data using conventions equivalent to the general-purpose registers. However, the floating-point instruction set provides no operations and no load/store instructions for these narrow types, and thus these cases are not described here.

Floating-point register pairs are shown with the lower-numbered register lower on the page.

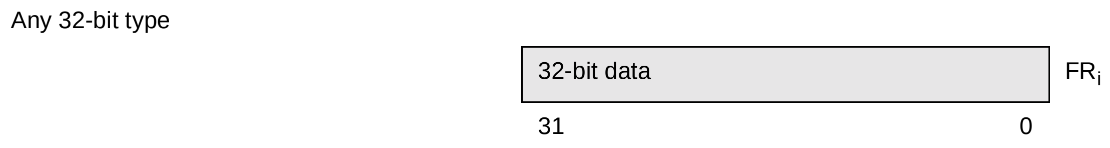


Figure 14: Data representation in single-precision registers (32-bit types)

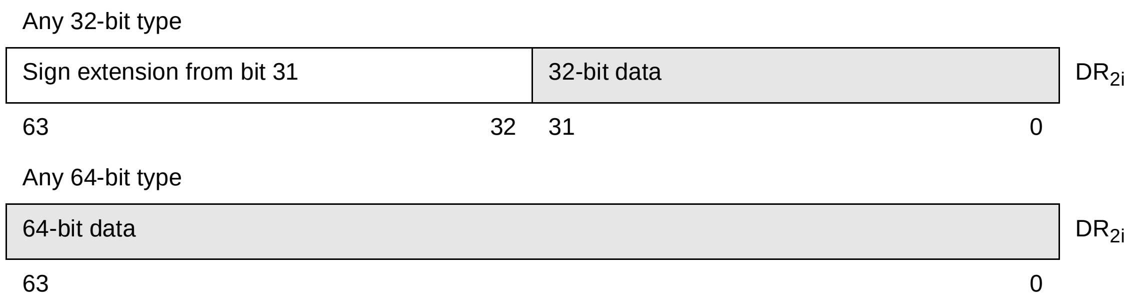


Figure 15: Data representation in double-precision registers (32-bit and 64-bit types)

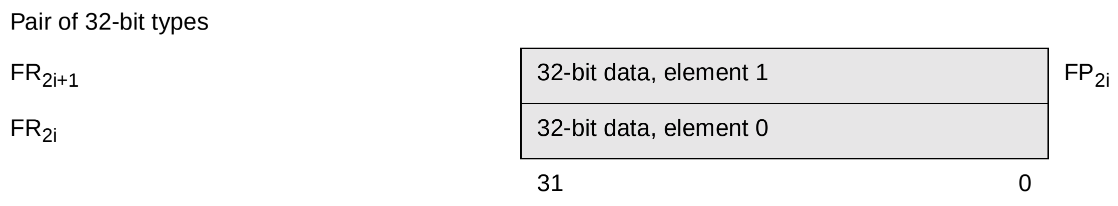


Figure 16: Data representation in pairs of single-precision registers (pair of 32-bit types)

3.7 Data representation in memory

Memory is byte-addressed and accessed using effective addresses generated by load and store instructions. The CPU uses 64-bit effective address arithmetic and this generates accesses into the effective address space in the inclusive range starting at byte number 0 and ending at byte number 2^{64}-1.

Address space is depicted with the lowest address lowest on the page, and the highest address highest on the page. Where memory diagrams are drawn with the memory element being larger than a byte, the memory element is depicted with the same left-to-right byte ordering conventions as used when depicting registers. This means that the byte numbering in this memory element varies according to endianness, but the bit numbering (and the depiction of the held value) is unchanged.

The CPU directly supports naturally aligned access. An object is naturally aligned if its address in memory is an exact integral multiple of its size.

The CPU supports accesses to objects that are not naturally aligned using special load and store instructions. These instructions synthesize a misaligned access using two aligned accesses plus appropriate masking and shifting.

The data layout in memory is determined by the width of the data type and the endianness of the CPU. The width of the data type determines the number of bytes in the memory representation. Unused bytes in the register representation are not held in the memory representation. The byte labeling within the register depends on endianness, and this determines the byte ordering in memory.

The mappings for values held in general-purpose registers, for single-precision floating-point registers and for double-precision floating-point registers are straightforward. These values each consist of a single object which is mapped into memory, as dictated by endianness, in a consistent and obvious manner.

The mapping of pairs of single-precision floating-point registers differs because a pair of values must be treated as two objects which are mapped consecutively into memory, with endianness ordering applied to each separate object in turn. The lower-numbered register within the register pair is mapped into lower memory addresses than the higher-numbered register. This allows the natural array ordering to be maintained as these values are moved between registers and memory.

The mappings for little endian and big endian memory organizations are illustrated in the following diagrams. In each case, a data type of a particular width in a register is shown (on the left hand side), along with the memory representation of that data type when held in memory at address A (on the right hand side).

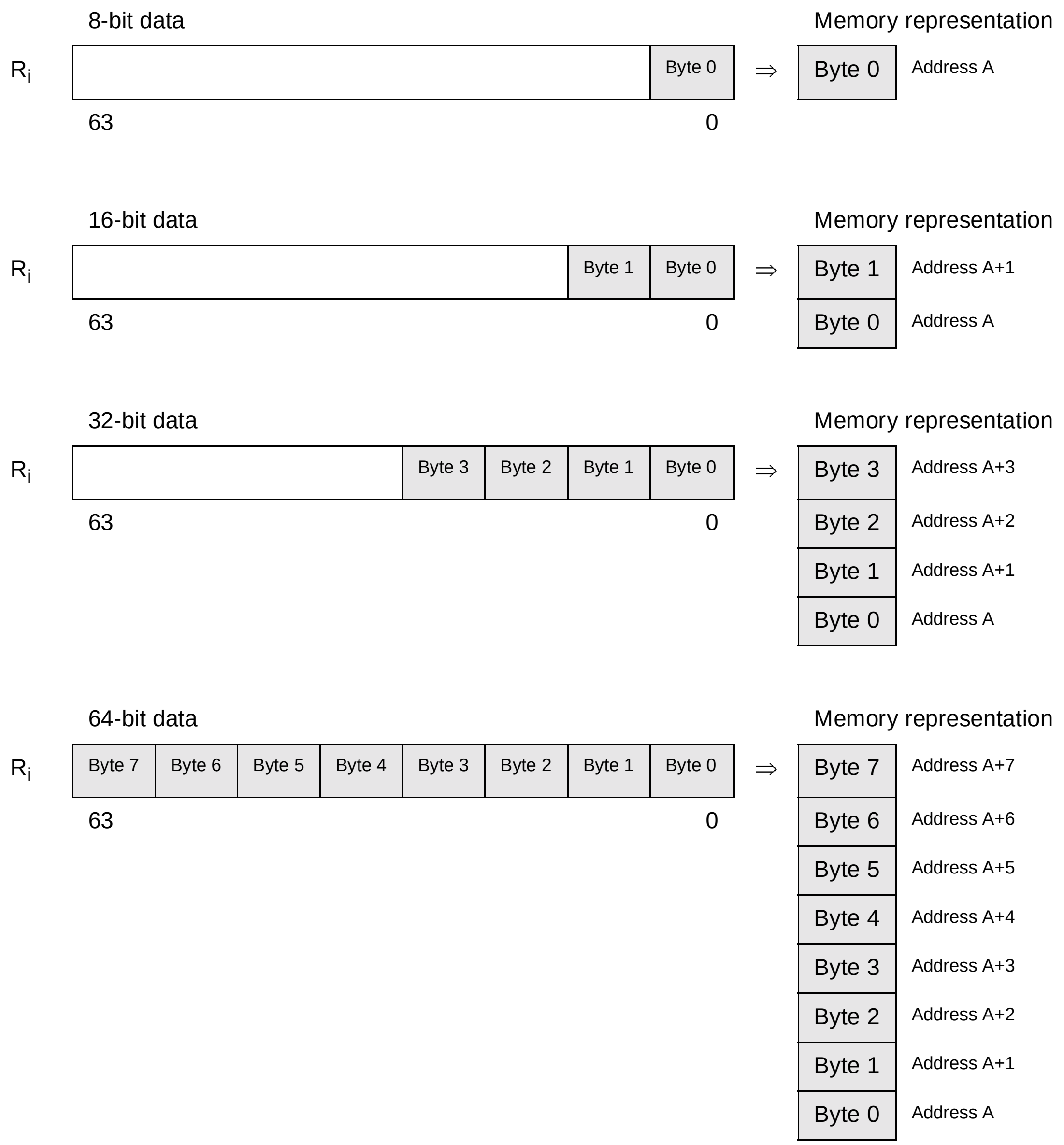


Figure 17: Little endian memory representation of values in general-purpose registers

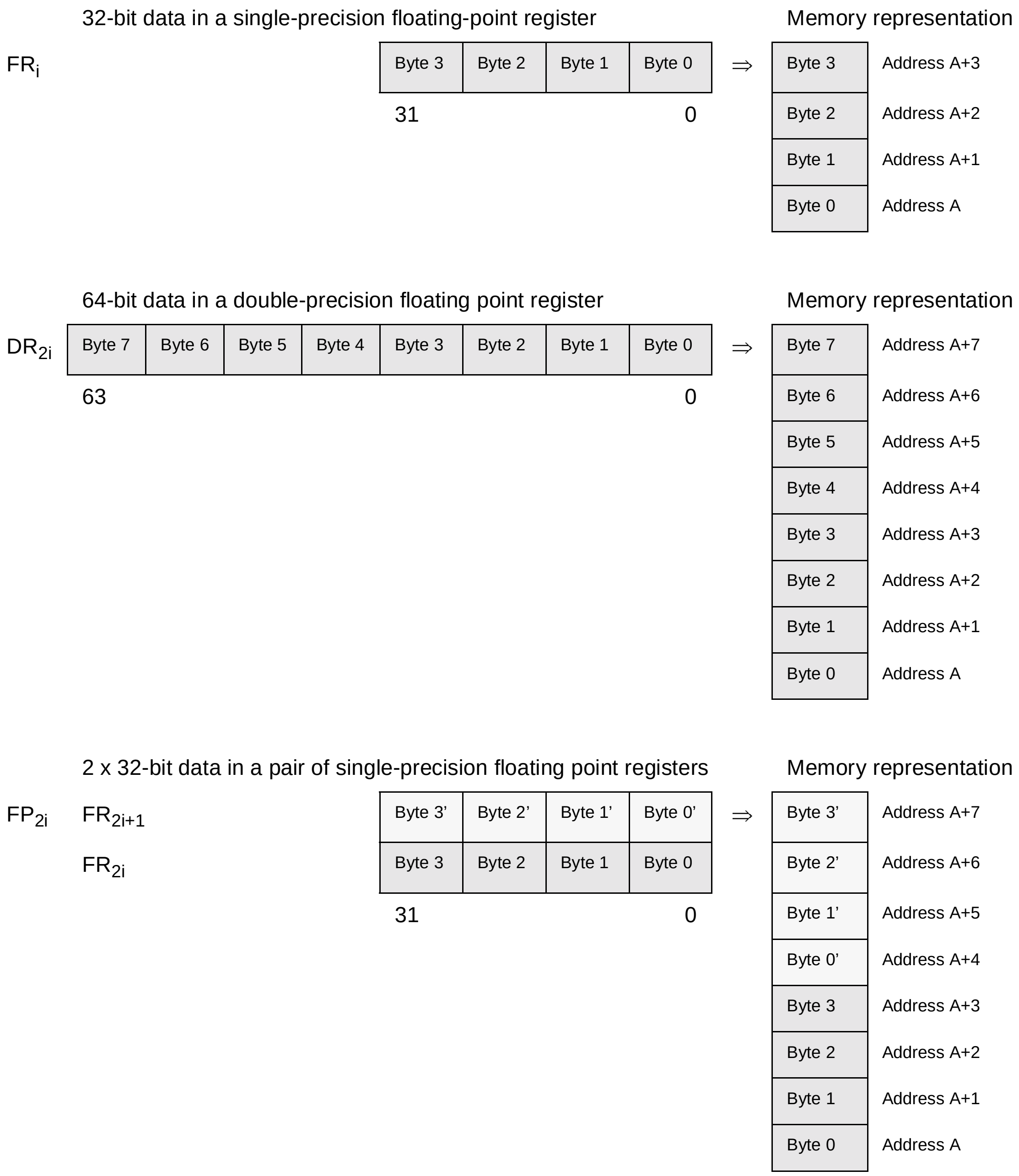


Figure 18: Little endian memory representation of values in floating-point registers

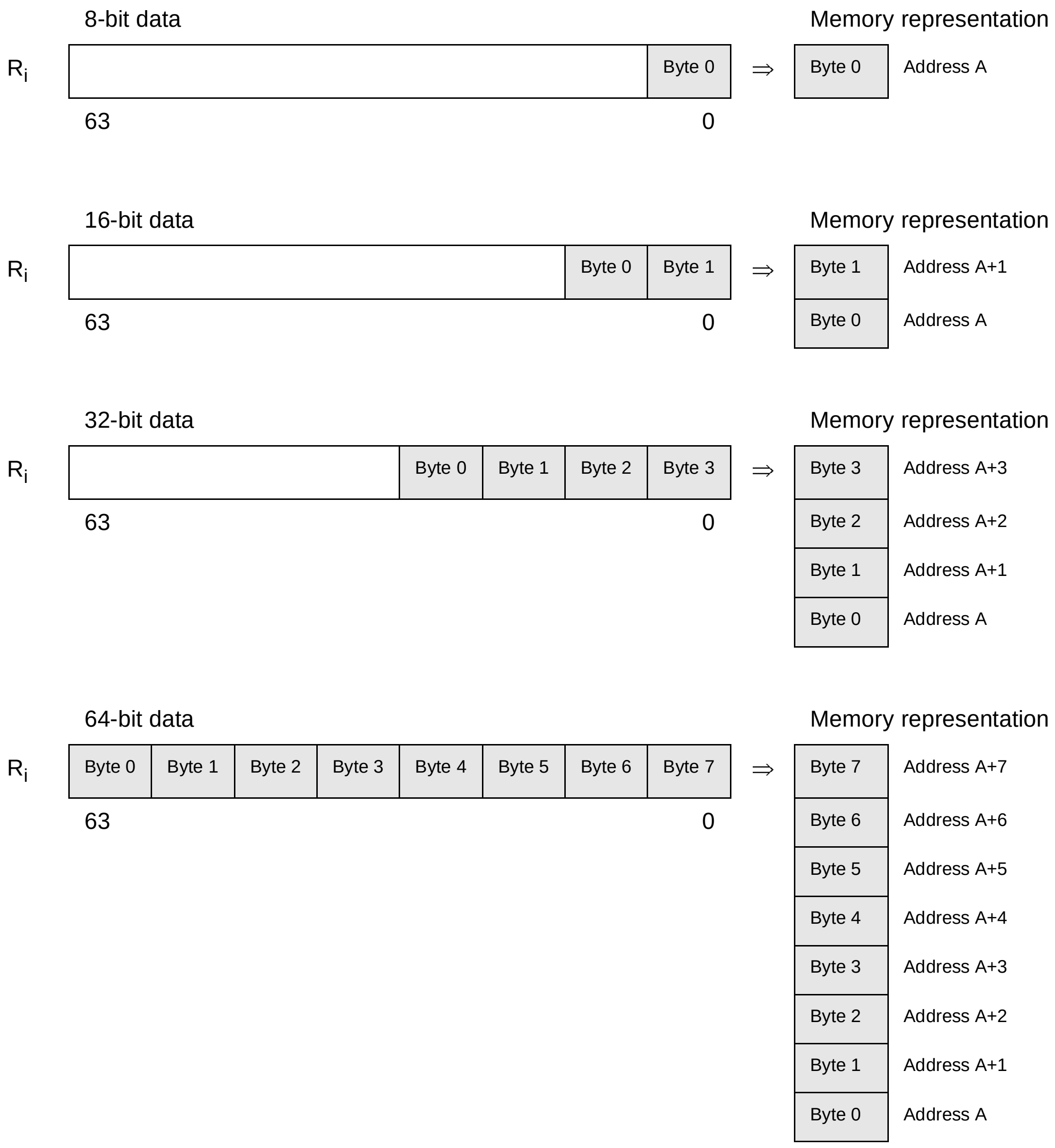


Figure 19: Big endian memory representation of values in general-purpose registers

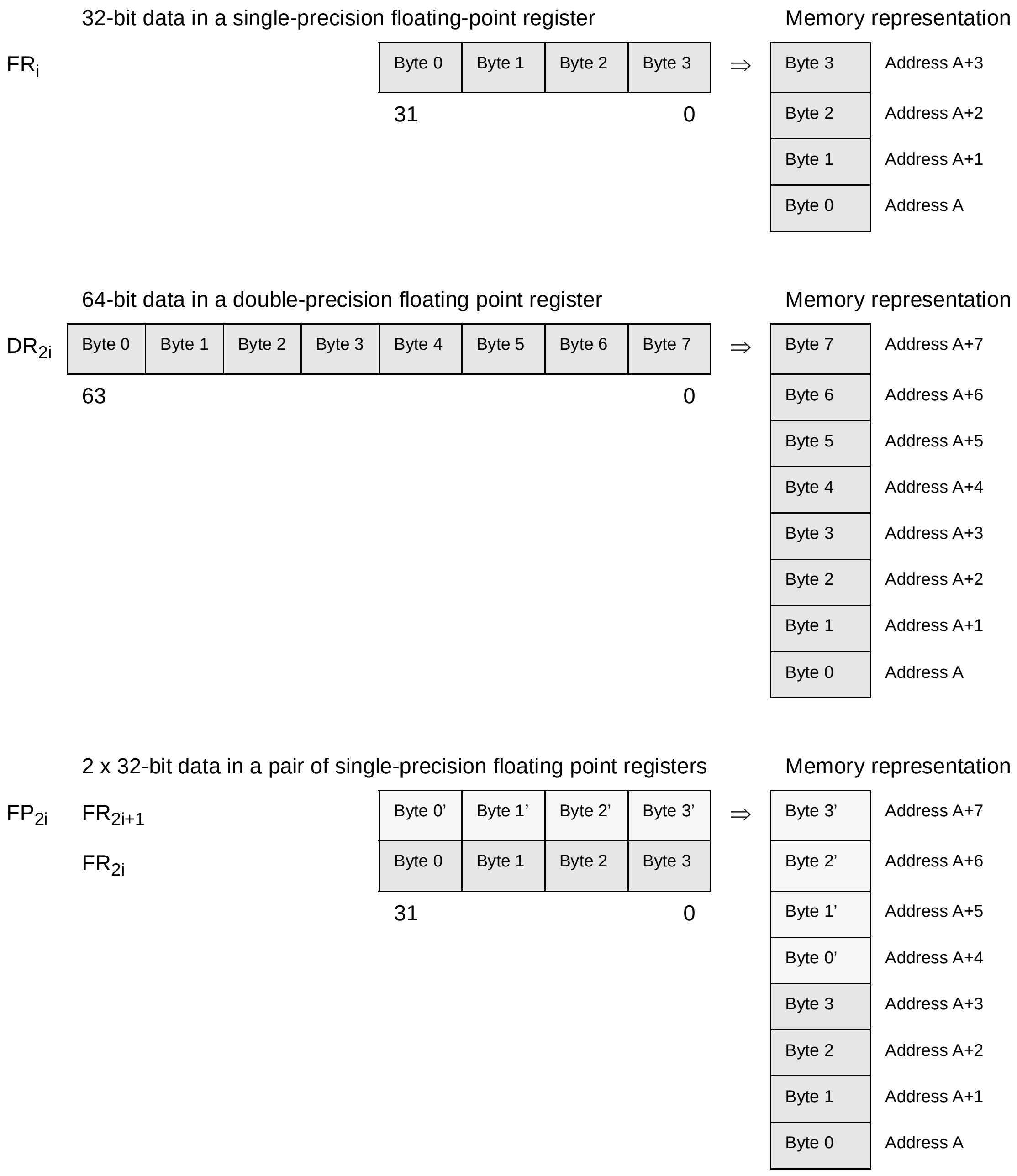


Figure 20: Big endian memory representation of values in floating-point registers

The memory mapping of a pair of single-precision registers is not always the same as that of a double-precision register:

- Regardless of endianness, the lower single-precision register of a pair is mapped into lower memory addresses than the upper single-precision register.
- A double-precision register must be mapped into memory as one object to give the correct representation for both endiannesses. Additionally, a double-precision register can also be viewed as two single-precision registers. The lower-numbered single-precision register contains the high part of the double-precision value and the higher-numbered single-precision register contains the low part. In order to give a consistent endianness-correct representation of the double-precision register, the mapping of these single-precision registers into memory must vary with endianness.

Thus the memory ordering of the two halves of a single-precision pair is independent of endianness, while the memory ordering of the two halves of a double-precision register varies with endianness.

It turns out that the memory mapping of single-precision pairs and that of double-precision registers is exactly the same in big endian mode, but different in little endian mode. This bias arises because the architecture always uses a big endian convention for splitting its double-precision registers into single-precision registers.

It is considered very bad practice for software to exploit these properties. The architecture strongly recommends that appropriate instruction sequences are always used:

- In SHmedia, the architecture provides a comprehensive set of load/store instructions. In particular, there are different instructions for the load/store of pairs of single-precision registers and for the load/store of double-precision registers. These instructions give the appropriate memory representations according to the endianness of operation.
- In SHcompact, there are instructions for the load/store of pairs of single-precision registers but *no* instructions for the load/store of double-precision registers. The load/store of double-precision registers should instead be synthesized using multiple single-precision load/store instructions. These sequences should be selected to give the appropriate memory representations according to the endianness of operation.

3.8 Effective address representation

The CPU supports a 64-bit effective address space. Effective addresses are unsigned quantities, giving the property that effective address 0 has the lowest address value.

An implementation of the CPU architecture implements all of the 64-bit effective address space, or a subset of the 64-bit effective address space. This is characterized by a single value, say neff, which represents the number of implemented bits of the effective address. These neff bits are always the least significant neff bits in the effective address. The architecture guarantees that neff will be at least 32 and at most 64 for any implementation.

The value of neff is used to size the implemented part of architectural state that contains effective addresses:

- Registers that hold effective addresses of instructions:
  - The program counter (PC).
  - Target registers (TR).
  - The following control registers: SPC, PSPC, RESVEC and VBR.
- Registers that hold effective addresses of data:
  - The following control register: TEA.

These registers contain neff implemented bits. Upper unused bits must always be a sign extension of the highest implemented bit (that is, bit neff-1), in order to be valid effective addresses.

General purpose registers contain 64 bits and can hold any address in the 64-bit effective address space. Programs should be restrained to the implemented part of the effective address space through software convention. The CPU causes instructions that access invalid effective addresses to take an exception.

A pictorial representation of the effective address space (not to scale) is shown in *Figure 21* for any valid neff, and in *Figure 22* for the special case of neff=32. Note that if neff is 64, then all of the effective address space is valid. In these diagrams, the shaded parts represent the valid effective address ranges. Also, the address labels show the address of the byte immediately above the adjacent horizontal line.

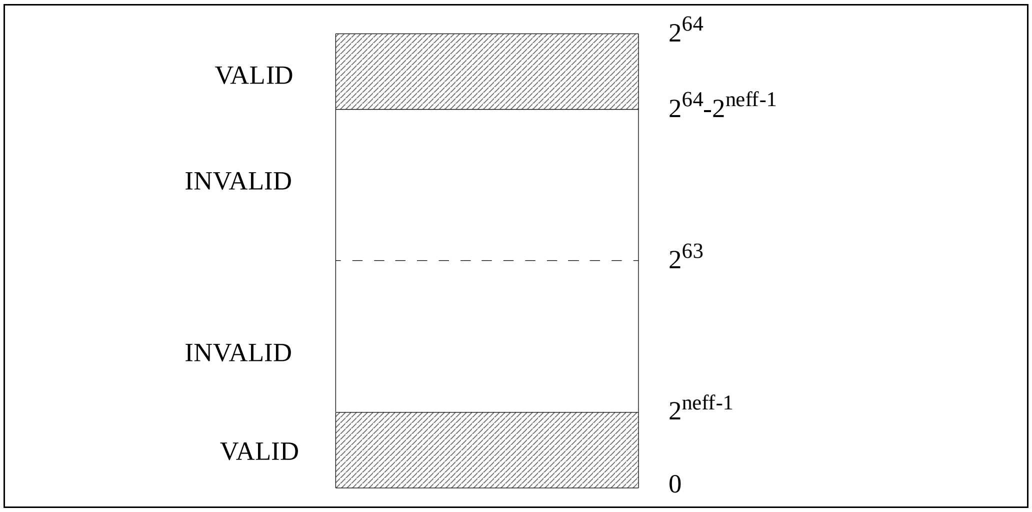


Figure 21: Effective address space for any valid neff

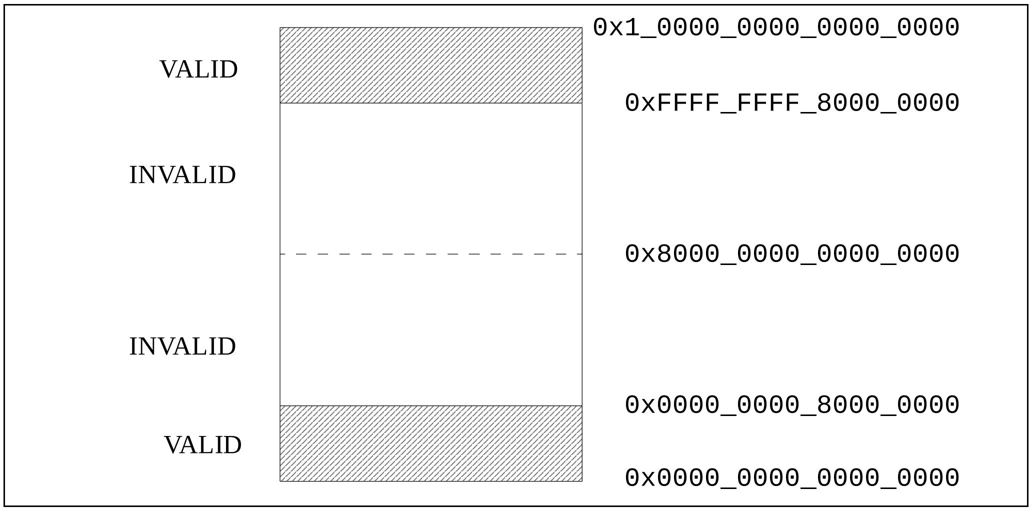


Figure 22: Effective address space for neff=32

3.9 Program counter overflow

The architecture is arranged so that a branch can never be taken to a program counter that is outside of the implemented part of the effective address space. This is because instructions that calculate a branch target address outside of this space raise an exception. Program counter (PC) overflow can occur, however, when instructions are executed near the upper limits of the implemented part of the effective address space.

PC overflow occurs when any instruction is executed and one of the following conditions is satisfied:

1 PC = 2^{64} - 2

2 PC = 2^{64} - 4

3 (PC = 2^{neff-1} - 2) AND (neff $\neq$ 64)

4 (PC = 2^{neff-1} - 4) AND (neff $\neq$ 64)

where neff is the number of implemented bits in an effective address.

Cases *1* and *3* can only occur for SHcompact instructions, while cases *2* and *4* can occur for both SHmedia and SHcompact instructions. If an implementation provides all of the effective address space then neff=64 and cases *3* and *4* do not exist.

PC overflow results in architecturally undefined behavior and must be avoided by software. Typically, an address map convention is used to prevent the execution of instructions near these upper limits.

3.10 Pointer representation

The choice between an unsigned pointer representation and a signed pointer representation depends upon software convention. This only affects whether pointer comparisons are implemented using unsigned or signed compares. Although the choice is not mandated by the architecture, the preferred representation is unsigned since this matches the convention for effective address space.

Software can choose between a 64-bit and 32-bit pointer representation.

64-bit pointer representation

For software using a 64-bit pointer, pointer values can reference any effective address. This includes all implemented and any unimplemented effective addresses. An access to any unimplemented effective address will result in an exception.

32-bit pointer representation

For software using a 32-bit pointer, the pointer should be held using a sign-extended 32-bit representation when held in registers (see *Figure 12*). Note that this sign-extended convention is used, regardless of whether software chooses to use a signed or unsigned pointer representation. The sign-extension of all 32-bit pointers matches the provision of a sign-extended effective address space. One half of the 32-bit pointer space maps to the bottom 2^{31} bytes of effective address space, and the other half maps to the top 2^{31} bytes of effective address space.

In fact, all values of 32-bit pointers correspond to effective addresses that will be implemented by all implementations. However, it is still possible for memory accesses in load and store instructions to overflow the sign-extended 32-bit space, because effective address calculation is performed at 64-bit precision. An access to any unimplemented effective address will result in an exception.

Invalid effective addresses

In both cases, software is expected to avoid invalid effective addresses. This could be achieved, for example, through software address map conventions. Accesses to invalid effective addresses could indicate of a programming error or of a program that needs more effective address than that provided by the implementation.

Effective addresses are mapped onto the physical address space of the machine by a memory management unit. The properties of this mapping are described in *Chapter 17: Memory management on page 271*

3.11 Other register representations

The CPU supports control and configuration registers. These registers are 64 bits wide. They are always accessed at this width and these accesses are independent of endianness. The details of their contents varies considerably from instance to instance. The standards used to specify these registers are described here.

The term ‘register’ in the following description refers to a control register or a configuration register only, but not to other kinds of register.

3.11.1 Register naming

Each register has a unique name. Register names are composed hierarchically by concatenating sub-names together separated by a period (‘.’). Successive sub-names repeatedly refine the classification of the register. A register can be refined to a field by concatenating the register name with the field name separated by a period (‘.’). A field can be refined to a single bit by concatenating the field with the bit name separated by a period (‘.’).

The semantics of a register are, in general, specific to that register. However, there are conventions which all registers adhere to. These conventions are described in the following sections.

3.11.2 Register conventions

Each register is classified as either UNDEFINED or DEFINED.

UNDEFINED registers

UNDEFINED registers are used to reserve registers for future implementations. The behavior of accesses to UNDEFINED registers is not defined by the architecture. A read from an UNDEFINED register returns an architecturally-undefined value. A write to an UNDEFINED register leads to behavior that is architecturally undefined.

If a register is UNDEFINED, it is possible that this register could become DEFINED in a future implementation and exhibit a well-defined behavior. Software must neither read nor write UNDEFINED registers to allow portability to future implementations.

DEFINED registers

A DEFINED register is composed of one or more fields. Each field is a contiguous collection of bits in the register. All bits in a DEFINED register belong to a field. Further categorization of a DEFINED register is performed at the field level.

3.11.3 Field conventions

Each field in a DEFINED register is classified as one of RESERVED, EXPANSION, READ-ONLY, READ-WRITE or OTHER.

In addition to the above defined field types, a field is also either volatile or non-volatile. A non-volatile field is not changed autonomously by hardware (excluding reset sequences), while a volatile field can be changed autonomously by hardware. When a field is volatile, the register specification describes the circumstances that cause the value to be autonomously modified.

When the value of a field is not architecturally defined, the field is said to have an undefined value. Many fields have an undefined value after power is first applied to the CPU.

A field can have some values which are reserved. These values must not be written into that field, otherwise the behavior of the access is not defined by the architecture. The specification of a particular writable field which has reserved values will enumerate the reserved values. It is possible that all values apart from one specific value could be reserved. In this case, the field must be programmed with only that specific value.

The field types are summarized in the following table.

Field type	Abbreviation	Usage
RESERVED	RES	Field is reserved
EXPANSION	EXP	Field is reserved for address space expansion
READ-ONLY	RO	Field is read-only and cannot be modified by software
READ-WRITE	RW	Field is readable and writable by software
OTHER	OTHER	Field has unusual semantics

Table 13: Register field types

RESERVED fields

RESERVED fields are used to reserve parts of a register for future expansion of the architecture. A read from a RESERVED field returns a zero. Writes to a RESERVED field are ignored. If a field is RESERVED, it is possible that this field will have a different behavior in a future implementation.

When reading from a control register, software should not interpret the value of any RESERVED fields. When writing to a control register with RESERVED fields, software should write these fields using a value previously read from that register. If no appropriate previous value is available, then software should write 0 to RESERVED fields. This approach will improve software portability to future implementations.

EXPANSION fields

EXPANSION fields are used to reserve parts of a register for future expansion of the address space. A read from a EXPANSION field returns a sign-extension of the highest implemented bit of the register. Writes to a EXPANSION field are ignored. Bits in EXPANSION fields can be used on future implementations to expand the address space using a sign-extended convention.

Software should always write a sign-extension of the highest implemented bit of the register into this field. This approach is necessary if software is to be executed on a future implementation with more implemented address space.

READ-ONLY fields

The value of a READ-ONLY field cannot be changed by software. A read returns the value associated with the field, while a write is ignored. A non-volatile READ-ONLY field has an immutable value.

READ-WRITE fields

A READ-WRITE field has conventional read and write behavior. A read returns the value of the field, while a write sets the value of the field.

OTHER fields

An OTHER field indicates that the field has unusual semantics. The specification of an OTHER field will describe the actual semantics.

4 SHmedia instructions

4.1 Overview

All SHmedia instructions are 4 bytes in length, and are held in memory on 4-byte boundaries. Instructions are described as collections of 32 bits, numbered from 0 (the least significant bit) to 31 (the most significant bit). The endianness of instructions in memory is dictated by the endianness of the processor.

If the processor is little endian, instructions are held in little-endian order in memory (see *Figure 23*). The least significant byte of an instruction, containing bits 0 to 7 of its encoding, is held at the lowest address in the memory representation (at address A). The most significant byte of this instruction, containing bits 24 to 31 of its encoding, is held at the highest address (at address A+3).

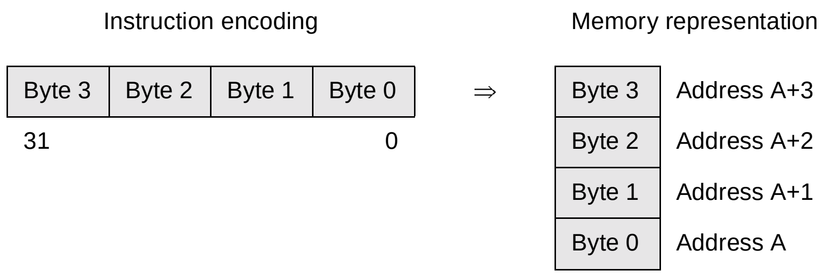


Figure 23: Little-endian memory representation of an SHmedia instruction

Alternatively, if the processor is big endian instructions are held in big-endian order in memory (see *Figure 24*). The most significant byte of an instruction, containing bits 24 to 31 of its encoding, is held at the lowest address in the memory representation (at address A). The least significant byte of this instruction, containing bits 0 to 7 of its encoding, is held at the highest address (at address A+3).

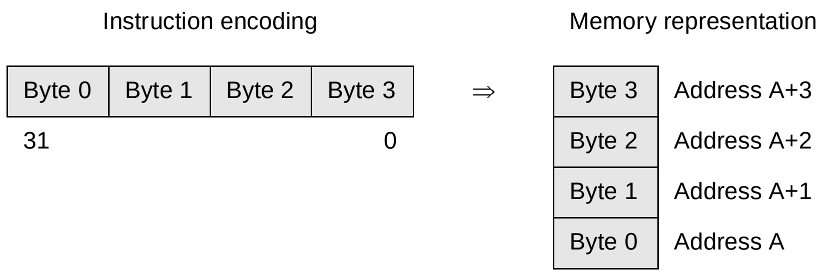


Figure 24: Big-endian memory representation of an SHmedia instruction

The following chapters (*Chapter 5: SHmedia integer instructions on page 69* to *Chapter 9: SHmedia system instructions on page 163*) summarize the SHmedia instruction set. Further details can be found in *Volume 2 Chapter 2: SHmedia instruction set*.

4.2 Instruction naming conventions

Each instruction mnemonic is formed from a basename followed by an optional modifier. A basename is a sequence of upper case letters and numbers.

The basename is an abbreviation or acronym related to the behavior of the instruction. The following conventions are used:

- All floating-point basenames start with ‘F’.
- All multimedia basenames start with ‘M’.
- Immediate is abbreviated to ‘I’; dynamic is abbreviated to ‘D’.
- High is abbreviated to ‘HI’; low is abbreviated to ‘LO’.
- Indexing is abbreviated to ‘X’.
- Saturation is abbreviated to ‘S’.

4.2.1 Type modifiers

The type modifier indicates the width and sign of the operation. A type modifier is a ‘.’ character followed by a sequence of upper case letters. If there is no type modifier then the operation has either register width or has no associated width. The basic modifiers are shown in *Table 14*.

Modifier	Width of operation
.B	byte, 8 bits (signed)
.UB	unsigned byte, 8 bits
.W	word, 16 bits (signed)
.UW	unsigned word, 16 bits
.L	long-word, 32 bits
.Q	quad-word, 64 bits
.S	single-precision floating-point, 32 bits
.P	pair of single-precision floating-point, 2 x 32 bits
.D	double-precision floating-point, 64 bits

Table 14: Basic type modifiers

Some instructions have different widths associated with source and destination values. These use the compound type modifiers shown in *Table 15*.

Modifier	Source	Destination	Modifier	Source	Destination
.DL	double-precision	long-word	.SD	single-precision	double-precision
.DQ	double-precision	quad-word	.SL	single-precision	long-word
.DS	double-precision	single-precision	.SQ	single-precision	quad-word
.LD	long-word	double-precision	.UBQ	unsigned byte	quad-word
.LS	long-word	single-precision	.WB	word	byte
.LW	long-word	word	.WL	word	long-word
.QD	quad-word	double-precision	.WQ	word	quad-word
.QS	quad-word	single-precision	.WUB	word	unsigned byte

Table 15: Compound type modifiers

4.2.2 Hint modifiers

Hint modifiers are used to indicate performance hints for control flow instructions. A hint modifier is a '/' character followed by an upper case letter. The interpretation placed on this information is described in *Section 5.3: Control flow instructions on page 72*. The available hint modifiers are shown in *Table 16*.

Modifier	Performance hint
/L	Likely hint
/U	Unlikely hint
(none)	Defaults to a likely hint

Table 16: Hint modifiers

4.3 Format conventions

Every instruction is associated with an instruction format. The format of an instruction determines how that instruction is encoded and decoded.

4.3.1 Format bit-fields

Each instruction contains 32 bits. These bits are grouped into contiguous collections of bits, termed a bit-field. Each bit-field is associated with a bit-field type. The available types are denoted by single character identifiers and are listed in *Table 17*.

Bit-field type	Bit positions	Bit-field size	Interpretation of the bit-field	Extracted as:
o	[26, 31]	6	Opcode (all instructions)	Unsigned
e	[16, 19]	4	Extension opcode (for some instructions)	Unsigned
r	Varies	Varies	Reserved (see *Section 4.5*)	Not applicable
x	Varies	6	Unused operand (see *Section 4.5*)	Not applicable

Table 17: Format bit-field conventions

Bit-field type	**Bit positions**	**Bit-field size**	**Interpretation of the bit-field**	**Extracted as:**
m	[20, 25]	6	General-purpose register (left source)	Unsigned
n	[10, 15]	6	General-purpose register (right source)	Unsigned
d	[4, 9]	6	General-purpose register (destination)	Unsigned
y	[4, 9]	6	General-purpose register (extra source)	Unsigned
w	[4, 9]	6	General-purpose register (source and destination)	Unsigned
b	[20, 22]	3	Target address register (left source)	Unsigned
c	[4, 6]	3	Target address register (extra source)	Unsigned
a	[4, 6]	3	Target address register (destination)	Unsigned
l	9	1	Likely bit (this is a performance hint for control flow)	Unsigned
g	[20, 25]	6	Floating-point register (left source)	Unsigned
h	[10, 15]	6	Floating-point register (right source)	Unsigned
f	[4, 9]	6	Floating-point register (destination)	Unsigned
z	[4, 9]	6	Floating-point register (extra source)	Unsigned
q	[4, 9]	6	Floating-point register (source and destination)	Unsigned
k	[20, 25]	6	Control register (left source)	Unsigned
j	[4, 9]	6	Control register (destination)	Unsigned
i_n	[10, 9+n]	n	n-bit unsigned immediate constant (right source)	Unsigned
s_n	[10, 9+n]	n	n-bit signed immediate constant (right source)	Signed

Table 17: Format bit-field conventions

For each bit-field type, this table shows:

- the bit positions in the format used to encode that bit-field. Reserved and unused bits can occur in a variety of bit positions. For immediates, the start bit position is constant but the range of bit positions depends upon the size of the immediate. The bit positions associated with all other bit-field types are constant.
- the interpretation placed on the bits of that bit-field. This determines, for example, whether that bit-field is used as an opcode, to identify a register, to encode an immediate value or is reserved.

- whether the value of that bit-field is extracted from the instruction encoding as an unsigned or as a signed value.

The bit-field type determines how to encode/decode that bit-field, and what the bit-field is used for. These properties apply to any instruction that contains that bit-field type in its format.

All formats have an opcode bit-field and this occupies the same bit positions in all formats. This arrangement allows the instruction decoder to determine the format quickly for any instruction, impose bit-field boundaries and decode the other bit-fields. Some formats also have an extension opcode bit-field to allow more instructions to be encoded.

Bit-fields in an instruction are encoded using the obvious binary representation. The highest bit number in the bit-field is most-significant; the lowest bit number in the bit-field is least significant.

A bit-field is extracted as either unsigned or signed as indicated by the bit-field type. When an n-bit unsigned bit-field is extracted from an encoding, it is zero-extended to give an integer in the range $[0, 2^n)$. When a n-bit signed bit-field is extracted from an encoding, it is sign-extended to give an integer in the range $[-2^{n-1}, 2^{n-1})$. Only signed immediate constants are extracted as signed numbers; all other bit-fields are extracted as unsigned numbers.

4.3.2 Major and minor formats

The formats are described using a two level scheme consisting of major and minor formats. Each major format has a different bit-field layout. The number of major formats is minimized to ease decoding.

Each major format is associated with a set of minor formats. All minor formats that are associated with a particular major format have the same bit-field layout. Minor formats are distinguished from each other because they have different operand types and interpret the operands in different ways. For example, a bit-field encoding a register operand in a major format, could be interpreted as a general-purpose register or a floating-point register in different minor formats.

Many minor formats are distinguished in order to allow operand type information to be determined early in instruction decode.

4.3.3 Format names

Many of the bit-field types represent operand details such as register numbers or immediate constants. Names for minor formats are systematically constructed by concatenating the bit-field types corresponding to the used operands, and using ‘x’ for unused operands. The order of concatenation is the order in which the operands are seen in a scan from most significant bit to least significant bit. The number of bits in the immediate field is appended, or ‘0’ if no immediate.

For example, the format ‘mnd0’ is used for instructions where the first operand is a general-purpose register (left source), the second operand is a general-purpose register (right source), and the third operand is a general-purpose register (destination). The format ‘xsd16’ is used for instructions where the first operand is unused (no left source), the second operand is a signed 16-bit immediate (right source), and the third operand is a general-purpose register (destination).

Major formats are named in a similar way to minor formats, but using capitalized versions of the bit-field types. Additionally, the major format names abstract away from the types of register operand, and only draw the following distinctions:

- ‘M’ indicates that the first operand is a register.
- ‘N’ indicates that the second operand is a register.
- ‘D’ indicates that the third operand is a register.
- ‘S’ indicates that the second operand is an immediate.
- ‘X’ indicates that the operand is unused.

4.4 Major formats

A summary of the major formats is given in *Table 18*. For each format this gives the interpretation of the 3 operands. Here, a 'register' operand could be a general-purpose register, target address register, floating-point register or control register. Not every instruction uses all of the operands provided by its major format.

Format Name	Operand 1	Operand 2	Operand 3
MND0	register	register	register
MSD6	register	sign-extended 6-bit immediate	register
MSD10	register	sign-extended 10-bit immediate	register
XSD16	never used	sign-extended 16-bit immediate	register

Table 18: Major format summary

The bit-field layout of the major formats is given in the following diagrams.

MND0

opcode	register	ext	register	register	reserved
31 26	25 20	19 16	15 10	9 4	3 0

MSD6

opcode	register	ext	6bit immediate	register	reserved
31 26	25 20	19 16	15 10	9 4	3 0

MSD10

opcode	register	10 bit immediate	register	reserved
31 26	25 20	19 10	9 4	3 0

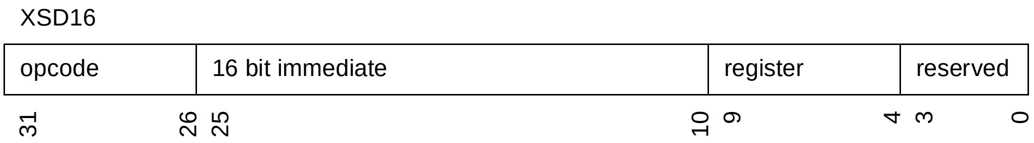


Figure 25: Major instruction formats

Volume 2, Appendix A: SHmedia instruction encoding describes minor formats and opcode assignments.

4.5 Reserved bits

The architecture requires that reserved bits in the instruction encodings are set to specific values. This allows future expansion of the instruction set without invalidating existing binaries. The use of inappropriate values leads to a reserved instruction exception (see *Table 19*) or to architecturally-undefined behavior (see *Table 20*).

Bits [0,3] of every instruction are reserved for the future expansion of the instruction set architecture. They must be set to 0b0000 on all instructions. In the current architecture specification, execution of an instruction with a non-zero value in bits [0,3] leads to a reserved instruction exception. This exception check is performed prior to decoding the opcode and extension opcode of the instruction.

Software should not rely on the reserved instruction exception generated for incorrect settings of bits [0, 3]. On a future implementation, the behavior for non-zero values could be modified to add new mechanism to the architecture.

Bit-field Type	Reserved bits	Required value	Behavior for inappropriate value
r	Bits [0,3] of every instruction	0b0000	Reserved instruction exception

Table 19: Reserved encoding fields (architecturally-defined behavior)

Unused opcode and extension opcode values are reserved for future expansion of the instruction set. Execution of a reserved instruction opcode leads to an exception, as described in *Volume 2, Appendix A: SHmedia instruction encoding*.

Reserved bits in a used operand field must be encoded as zero or the behavior is architecturally undefined. An operand field contains unused bits in the following cases:

- Bits 3, 4 and 5 of a left source operand for a target address register.
- Bits 3 and 4 of an extra source operand or destination operand for a target address register. Bit 5 is the 'likely' bit in these cases.
- Bit 0 of an operand for a double-precision floating-point register.
- Bit 0 of an operand for a single-precision floating-point register pair.
- Bits 0 and 1 of an operand for a floating-point register vector.
- Bits 0, 1, 2 and 3 of an operand for a floating-point register matrix.

Unused operands are marked as 'x' in minor format names. The required encoding for unused operands depends upon the operand position and type:

- If the first operand is unused, then it is an unused source.
- If the second operand is unused, then it is an unused source.
- If the third operand is unused, then it is an unused destination.

If the first operand is used to hold a floating-point register and the second operand is unused, then the second operand is also considered to be a floating-point operand. In all other cases, unused operands are considered to be general-purpose operands.

These distinctions are used in *Table 20* to determine how the unused operand should be encoded.

Bit-field Type	Reserved bits	Required value	Behavior for inappropriate value
r	Reserved bits in a used operand field	0	Architecturally undefined
x	Unused general-purpose source operand	0b111111 (this corresponds to a read of R63)	Architecturally undefined
	Unused general-purpose destination operand	0b111111 (this corresponds to a write to R63)	Architecturally undefined
	Unused floating-point source operand	Set to the same value as the used floating-point source operand	Architecturally undefined
	Unused floating-point destination operand	0b000000	Architecturally undefined

Table 20: Reserved encoding fields (architecturally-undefined behavior)

4.6 Assembly notation

This manual uses a straightforward assembly notation for describing example instruction sequences.

Each instruction consists of a mnemonic and up to three operands. Mnemonics correspond to the instruction names used in this manual, though they can be specified in either lower or upper case.

The operands are separated by commas. The order of the operands and their meaning are determined by the instruction. Each operand is either a register designator or an expression.

Register designators are the same as the architectural register names, except that the subscript notation is not used. For example, R0 is the assembly syntax used to represent R_0. Additionally, the register designators can be specified in either lower or upper case.

Expressions are constructed using standard integer operators and literal notation. In expressions, the symbol ‘$’ indicates the value of the current instruction’s PC.

Program labels are alphanumeric strings. The location of a label is defined by inserting the label name followed by a ':' in the text. The value of that label is referred to by quoting the label name in an expression.

For a label referring to an instruction, the value of the label is the absolute address of that instruction with the lowest bit indicating the instruction mode (0 for SHcompact, 1 for SHmedia). For a label referring to data, the value of the label is the absolute address of that data.

Comments are introduced by a ';' prefix and terminated by the end of the line. In the examples, portions of omitted code are indicated with a line beginning with '...'.

SHmedia integer instructions 5

5.1 Overview

The SHmedia instruction set provides efficient support for the most common integer operations found in typical programs. The integer instruction set contains the following groups of instructions: constant loading, control flow, arithmetic, comparison, bitwise operations, shifts and miscellany.

These instructions are described in this chapter.

5.1.1 Control flow

The performance of many programs is highly dependent on the efficiency of branches. The control flow mechanism has therefore been designed to support low-penalty branching.

This is achieved by allowing separation of the *prepare-target* instruction that notifies the CPU of the branch target, away from the *branch* instruction that causes control to flow, perhaps conditionally, to that branch target, This technique allows the hardware to be informed of branch targets many cycles in advance, enabling a smooth transition from the current sequence of instructions to the target sequence, should the branch be taken.

The arrangement also allows for more flexibility in the branch instructions, since the branches now have sufficient space to encode a comprehensive set of compare operations.

5.1.2 64-bit integer operations

The natural length of operation is 64 bits. General-purpose registers are 64 bits wide and effective address calculation is performed at 64-bit precision. The provision of 64-bit support allows the architecture to support programming models with 64-bit arithmetic and 64-bit addressing.

A sufficient set of instructions is provided to perform the most common integer operations with 64-bit data. The instructions include constant loading, comparison, addition, subtraction, bitwise operations and shifts.

Signed and unsigned 64-bit integers are supported. Some operations do not need to take into account the sign of the operands. In these cases a single instruction is provided which can be used on both signed and unsigned 64-bit integers.

5.1.3 32-bit integer operations

In many programs, the most common length of operation is 32 bits. Many of the common integer operations using 32-bit data can be mapped directly onto operations at 64-bit width. This includes constant loading, comparison, bitwise operations and some shifts; there is no need to include specialized versions of these instructions for 32-bit data. Other common integer instructions do require the narrower width of 32-bit data to be accounted for. Instructions are provided to perform addition, subtraction, multiplication and the remaining shifts with 32-bit data.

Signed and unsigned 32-bit integers are supported. The representations of these data types in registers treat the upper 32 unused bits in a consistent way. The upper 32 bits are held as sign extensions of bit 31 regardless of whether the register represents a signed or unsigned 32-bit integer. This is known as a 32-bit sign-extended representation.

The use of a consistent representation for signed and unsigned 32-bit integers simplifies interactions between SHmedia and SHcompact. In cases where the operation does not need to take into account the sign of the operands, this uniformity allows some instructions to be used on both signed and unsigned 32-bit integers.

5.1.4 Other integer operations

Some operations do not occur commonly enough in typical programs to justify support as a single instruction. Neither divide nor remainder instructions are supported, for example, and these must be implemented in software where required. A full set of multiplies is not supported either; the missing operations can be readily synthesized in sequences using the multiplies provided.

5.2 Constant loading instructions

Constant values are very common in typical programs. Many instructions have an immediate operand to allow a range of constant values to be encoded directly in the instruction. If the required constant does not fit, then the constant must be loaded separately. Some instructions do not have an immediate operand and cannot encode a constant directly. Again the constant must be loaded separately.

Two instructions are provided for loading constants. MOVI loads a register with sign-extended 16-bit immediate value. SHORI shifts its source operand 16 bits to the left, and then combines it with its 16-bit immediate value using a bitwise OR operation. Constants, up to 64 bits in length, can be loaded by using a MOVI instruction followed by zero or more SHORI instructions as required. Sign-extended 16-bit constants can be loaded in 1 instruction, sign-extended 32-bit constants in 2 instructions, sign-extended 48-bit constants in 3 instructions and 64-bit constants in 4 instructions.

Instruction	Summary
MOVI source,result	move immediate
SHORI source1,source2_result	shift then or immediate

Table 21: Constant loading instructions

Reads from R_{63} always returns zero, and writes to R_{63} are always ignored. This can be used to make any register operand take a zero value. This is very useful as zero is a particularly common constant. It can also be used to discard the result of an instruction.

5.3 Control flow instructions

The instructions that notify the CPU of the branch target are termed *prepare target* instructions, and use mnemonics that commence with the letters ‘PT’. The only required architectural effect of these instructions is to calculate the target address, raise an appropriate exception if this address is misformed or otherwise store the address in the specified target address register.

There are 8 target address registers. These registers are 64-bit wide, but implementations need only implement enough bits to allow representation of all valid target addresses.

The target address registers are written by prepare-target instructions. They are read by branch instructions and the GETTR instruction (see *Section 5.3.4: The GETTR instruction on page 79*). If a prepare-target instruction calculates a target address which is outside the implemented effective address space, then the instruction raises an exception. Thus, it is not possible for a target address register to hold an address outside of the implemented effective address space.

All SHmedia instructions are 4 bytes in size, and 4-byte aligned in memory. This means that there is no need to encode the lowest 2 bits of SHmedia instruction addresses. All SHcompact instructions are 2 bytes in size, and 2-byte aligned in memory. This means that there is no need to encode the lowest 1 bit of SHcompact instruction addresses.

The lowest bit of the target instruction address is used to distinguish the mode of the target instruction: SHmedia has the lowest bit set to 1, and SHcompact has it as 0. If a prepare-target is executed such that the bottom 2 bits of the target address are both set, then an exception is raised to signal a misaligned SHmedia instruction. The branch architecture is arranged so that the PC itself can never become misaligned.

Only the unconditional branch is capable of mode switching. Conditional branches disregard the value of the bottom 2 bits of the target address, and do not cause a mode switch. This arrangement permits simpler and more efficient implementations by limiting the mode switch mechanism to one SHmedia instruction.

Note that the misaligned SHmedia instruction check is performed at prepare-target time regardless of the nature of the branch. Thus, neither an unconditional nor a conditional branch can ever be executed with a target address register corresponding to a misaligned SHmedia instruction.

The interpretation of the four possible combinations of the lowest 2 bits are shown in *Table 22*. Note that not all prepare-target instructions are capable of generating all combinations.

Bit 1	Bit 0	Interpretation
0	0	Target is an SHcompact instruction on a 4-byte boundary
0	1	Target is an SHmedia instruction
1	0	Target is an SHcompact instruction on a 2-byte, but not on a 4-byte, boundary
1	1	Target is a misaligned SHmedia instruction, exception raised at prepare-target

Table 22: Interpretation of lowest 2 target address bits by an unconditional branch

Bit 1	Bit 0	Interpretation
0	0	Target is an SHmedia instruction Lowest 2 bits of the target address are discarded when forming the target PC
0	1	
1	0	
1	1	Target is a misaligned SHmedia instruction, exception raised at prepare-target

Table 23: Interpretation of lowest 2 target address bits by a conditional branch

5.3.1 Prepare-target instructions

Four different prepare-target instructions provide different ways of forming the target address.

Instruction	Summary
PTA offset,target	prepare target relative immediate (target indicates SHmedia)
PTB offset,target	prepare target relative immediate (target indicates SHcompact)
PTABS address,target	prepare target absolute register
PTREL offset,target	prepare target relative register

Table 24: Prepare-target instructions

The PTA instruction forms the target address by adding a constant value onto the PC of the current instruction. The constant is formed by taking a 16-bit immediate value, shifting it left by 2 bits and adding 1. The target address always indicates an SHmedia instruction. The target instruction has a byte displacement from the PTA instruction which is within the range [-131072, 131068]. This instruction is typically used for 'direct' branching to SHmedia instructions that are within range.

The PTB instruction forms the target address by adding a constant value onto the PC of the current instruction. The constant is formed by taking a 16-bit immediate value and shifting it left by 2 bits. The target address has the lowest bit clear. This will cause a switch to SHcompact if it is used by an unconditional branch; no mode switch occurs if it is used by a conditional branch. The target instruction has a byte displacement from the PTB instruction which is within the range [-131072, 131068].

The PTB instruction is typically used for 'direct' branching to SHcompact instructions that are within range. Note that bit 1 of the target address is always zero for PTB. This means that the target SHcompact instruction of a mode change using an immediate branch, must be 4-byte aligned. PTB can also be used, in conjunction with GETTR, for loading PC-relative addresses.

The PTABS instruction uses the value of the provided register as the target address. This can encode any of the 4 possible combinations of the lowest 2 bits. An instruction misalignment exception is raised if both of the lower bits are set. This instruction is typically used for 'indirect' branching, such as function return and call by function pointer.

The PTREL instruction forms the target address by adding a register value onto the PC of the current instruction. This can encode any of the 4 possible combinations of the lowest 2 bits. An instruction misalignment exception is raised if both of the lower bits of the target address are set. This instruction is typically used for 'direct' branching to either SHmedia or SHcompact instructions where the displacement is not within range of an immediate-based branch. In this case the long displacement can be loaded by a sequence of MOVI and SHORI instructions.

The placement of prepare-target instructions is expected to be highly optimized. Prepare-target instructions can be merged if they refer to the same target instruction. The main objective is to arrange the code so that each prepare-target instruction is maximally separated, in the dynamic instruction stream, from the branch instruction that reads that target address register. This gives the branch mechanism the maximum amount of time to arrange for the control flow to be achieved without penalty.

In practice, the migration of prepare-target instructions away from branches will be limited by the size of functions, by the finite number of target address registers and by data/control dependency (for indirect branches). The most important optimization is to hoist prepare-target instructions out of inner-most loops. Prepare-target optimizations can be achieved through standard compiler techniques such as loop/block-invariant code motion, common sub-expression elimination, register allocation and instruction scheduling.

Typically, many prepare-target instructions will be migrated to the beginning of the function. Most other prepare-target instructions will migrate to the beginning of basic blocks. Only a few prepare-targets will occur elsewhere, typically where their migration is blocked by a data or control dependency.

The encoding of all prepare-target instructions includes an l-bit. The l-bit ('l' for likely) should be used to indicate whether it is likely for control to be passed to that target address. If the instruction mnemonic has no modifier, or if it has a '/L' hint modifier, then the l-bit is encoded as 1. This indicates that it is considered likely for control to pass through to the branch instruction using that target address register and for that branch to be taken. If the instruction mnemonic has a '/U' modifier, then the l-bit is encoded as 0 indicating that this flow is considered unlikely.

The l-bit has no architectural effect on the behavior of the instruction, but is used to pass a performance hint to the implementation. An implementation can, for example, perform some prefetching of likely branch targets, but not of unlikely branch targets. An appropriate setting for this bit can be deduced, for example, from compiler heuristics or according to branch profiling information.

5.3.2 The unconditional branch instruction

One unconditional branch, BLINK, is provided.

Instruction	Summary
BLINK target,link	branch unconditionally and link

Table 25: Unconditional branch instruction

It is preferable to use this instruction for all unconditional branches, rather than a conditional branch with an always true condition. This allows the unconditional nature of the branch to be deduced without operand analysis.

BLINK writes the target address of the subsequent instruction to its destination register. The lowest bit of this target address is set to 1, indicating that this instruction is to be executed as an SHmedia instruction. This is a procedure link mechanism, since it allows the target instruction sequence to return control back to the instruction sequence that invoked it. This is typically used to implement standard call and return mechanisms.

The choice of link register is not dictated by the SHmedia instruction set. However, the SHcompact instruction set provides call and return instructions that use PR as the link register (see *Section 11.2: Control flow instructions on page 172*). PR is mapped into the SHmedia general-purpose register set as the lower 32 bits of R18 (see *Section 2.9: SHcompact state on page 26*). An SHmedia call sequence should therefore use R18 as the link register where interoperability with SHcompact procedures is required. Additionally, software must ensure that the SHmedia return address can be represented in the lower 32-bits of R18 (see *Volume 3, Chapter 1: SHcompact specification*).

The write to the destination register can be defeated using R63:

```
BLINK TRa, R63 ; transfer control to TRa without link
```

This can be used to achieve an unconditional branch without a link. Since BLINK is an unconditional branch it does not have a bit to indicate likelihood.

BLINK is the only SHmedia branch that can cause a mode switch. Bit 0 of the target address register indicates the target mode; 0 indicates SHcompact and 1 indicates SHmedia. Bit 1 indicates the alignment of the SHcompact target instruction (whether it is 4-byte aligned or not). Bit 1 is always 0 for SHmedia target instructions due to the prepare-target checks. BLINK should be used for function return since, in general, a function will not know whether it will be called from an SHmedia or an SHcompact caller.

5.3.3 Conditional branch instructions

There are six conditional branches that perform register with register comparisons.

Instruction	Summary
BEQ source1,source2,target	branch if equal 64-bit
BNE source1,source2,target	branch if not equal 64-bit
BGT source1,source2,target	branch if greater than 64-bit signed

Table 26: Conditional branch instructions

Instruction	Summary
BGE source1,source2,target	branch if greater than or equal 64-bit signed
BGTU source1,source2,target	branch if greater than 64-bit unsigned
BGEU source1,source2,target	branch if greater than or equal 64-bit unsigned

Table 26: Conditional branch instructions

The full set of register with register comparisons can be synthesized from these by swapping the source operands. This exploits the following two equivalences:

$(\mathbf{i} < \mathbf{j}) \equiv (\mathbf{j} > \mathbf{i})$
$(\mathbf{i} \leq \mathbf{j}) \equiv (\mathbf{j} \geq \mathbf{i})$

The provision of a full compare set means that the sense of the branch can be chosen arbitrarily. This is because the inversion of the branch condition can be folded, for free, into the compare using the following equivalences:

$\mathbf{NOT}\ (\mathbf{i} = \mathbf{j}) \equiv (\mathbf{i} \neq \mathbf{j})$
$\mathbf{NOT}\ (\mathbf{i} \neq \mathbf{j}) \equiv (\mathbf{i} = \mathbf{j})$
$\mathbf{NOT}\ (\mathbf{i} < \mathbf{j}) \equiv (\mathbf{i} \geq \mathbf{j})$
$\mathbf{NOT}\ (\mathbf{i} > \mathbf{j}) \equiv (\mathbf{j} \geq \mathbf{i})$
$\mathbf{NOT}\ (\mathbf{i} \leq \mathbf{j}) \equiv (\mathbf{i} > \mathbf{j})$
$\mathbf{NOT}\ (\mathbf{i} \geq \mathbf{j}) \equiv (\mathbf{j} > \mathbf{i})$

This flexibility allows the branch-taken instruction sequence and the branch-not-taken instruction sequence (that is, fall-through) to be swapped without any overhead. In general, the instruction sequences should be arranged to favor fall-through so as to avoid any branch penalty.

Branch conditions are often compares with zero. All cases can be represented in a single branch instruction, using the above instructions with one source operand set to R63.

There are two conditional branches that perform register with immediate comparisons.

Instruction	Summary
BEQI source1,source2,target	branch if equal to immediate 64-bit
BNEI source1,source2,target	branch if not equal to immediate 64-bit

Table 27: Conditional branch with immediate instructions

Only equality and inequality are provided for branch compares with immediates. Other forms require the immediate to be loaded separately. The provided pair are complements of each other, which again gives the property that the sense of the branch can be chosen arbitrarily.

Conditional branches are not capable of mode switch. Bits 0 and 1 of the target address are ignored.

The compare operations in the branches are performed on 64-bit integers. They are also directly usable for 32-bit integers, providing that the integers are held in a sign-extended 32-bit representation.

It is important to note that unsigned comparisons can be performed directly at 64-bit width on unsigned 32-bit numbers, even though those numbers are held in a 32-bit sign-extended representation. The effect of the sign-extension is to add (2^{64} - 2^{32}) onto the value of each of the unsigned 32-bit numbers in the range [2^{31}, 2^{32}). This translation from the unsigned 32-bit number space, via sign-extension, into an unsigned 64-bit number space preserves the ordering of all of these unsigned numbers. This means that unsigned 64-bit compares give the correct results.

All conditional branch instructions have an l-bit, which should be used to indicate whether that branch is likely to be taken. If the instruction mnemonic has no modifier, or if it has a '/L' modifier, then the l-bit is encoded as 1 indicating that the branch is considered likely to be taken. If the instruction mnemonic has a '/U' modifier, then the l-bit is encoded as 0 indicating that this branch is considered unlikely.

The l-bit has no architectural effect on the behavior of the instruction, but is used to pass a performance hint to the implementation. An implementation can use this information to indicate whether to favor execution down the taken or not-taken path. It can, for example, use a static prediction technique to begin execution of the predicted path before it is known whether the branch is actually taken or not. This can eliminate branch penalties, where the prediction is correct, but can incur branch penalties where the prediction is incorrect.

Appropriate setting of the l-bit is important for branch performance. An appropriate setting for this bit can be deduced, for example, from compiler heuristics or according to branch profiling information.

Note that l-bits are provided on both prepare-target and conditional branch instructions. The provided hints are similar but different. The prepare-target hint is typically used to control prefetch, while the branch hint is typically used to control prediction. The prepare-target hint has to factor in the likelihood of a branch instruction using that target address register being reached at all, whereas a branch hint simply distinguishes taken and not-taken.

Finally, there can be multiple branches associated with a prepare-target instruction, and the likelihoods of those different branches being taken can vary.

There are no exception cases associated with branch instructions.

5.3.4 The GETTR instruction

The GETTR instruction is used for copying a target address register to a general-purpose register.

Instruction	Summary
GETTR target,result	move from target register

Table 28: GETTR instruction

PTABS performs a complementary operation; it copies a general-purpose register into a target address register. The GETTR instruction always returns a value which can be reloaded into a target address register (using PTABS) without generating an exception. The value returned by GETTR ensures that any unimplemented higher bits of the source target register are seen as sign extensions of the highest implemented bit.

GETTR and PTABS can be used in sequences to save and restore target registers to and from memory. This can be used in calling conventions or in a context switch. GETTR can also be used for loading a PC-relative address. For example:

```
PTB label, TR0 ; PC-relative load of label into TR0
GETTR TR0, R0 ; move TR0 into R0
...
label: ; label should be 4-byte aligned
```

5.4 Arithmetic instructions

The arithmetic operations supported are addition, subtraction and multiplication. Neither divide nor remainder instructions are supported, and these must be implemented in software where required. The provided instructions use modulo arithmetic. Arithmetic overflow checking is not directly supported, and must be achieved with additional instructions where required.

Addition and subtraction are very common. Comprehensive support is provided for 32 bit and 64 bit operations, on signed and unsigned integers. In fact, the provided instructions do not need to take account of sign so only 4 instructions are needed.

Instruction	Summary
ADD source1,source2,result	add 64-bit
ADD.L source1,source2,result	add 32-bit
SUB source1,source2,result	subtract 64-bit
SUB.L source1,source2,result	subtract 32-bit

Table 29: Addition and subtraction instructions

ADD and SUB use all 64 bits of their sources and write the result to all 64 bits of their destination. ADD.L and SUB.L ignore the upper 32 bits of their sources, and write a 32-bit result that is sign-extended up to 64 bits. This arrangement means that ADD.L and SUB.L give a free narrowing type conversion on input, and always produce an output in a sign-extended 32-bit representation.

These instructions also have some other important uses. Negation can be achieved using the subtract operations with the first source being R63:

```
SUB R63, Rm, Rd ; 64-bit negation of Rm into Rd
SUB.L R63, Rm, Rd ; 32-bit negation of Rm into Rd
```

One way to sign extend a 32-bit value through the upper 32 bits is to use ADD.L with an R63 source:

```
ADD.L Rm, R63, Rd ; 32-bit sign extend of Rm into Rd
```

This is useful for converting from a 64-bit integer value down to a 32-bit integer value. This is appropriate for both signed and unsigned 32-bit numbers since the representation of both of these types is the same (sign-extended). Note that no instruction is required to convert between a signed 32-bit integer and an unsigned 32-bit integer since they have the same representation in registers.

Addition of a register value with a constant value, and subtraction of a constant value from a register value, are also directly supported. The constant is provided as a sign-extended immediate. This means that only add immediate need be provided. Subtract immediate can be achieved by adding the negation of the immediate. Separate instructions are provided for 32-bit and 64-bit operation, and these can be used on both signed and unsigned integers.

Instruction	Summary
ADDI source1,source2,result	add immediate 64-bit
ADDI.L source1,source2,result	add immediate 32-bit

Table 30: Addition with immediate instruction

ADDI uses all 64 bits of its register source and writes the result to all 64 bits of its destination. ADDI.L ignores the upper 32 bits of its register source, and writes a 32-bit result that is sign-extended up to 64 bits. This arrangement means that ADD.L gives a free narrowing type conversion on input, and always produces an output in a sign-extended 32-bit representation.

An instruction is provided to perform a 32-bit addition and zero-extend the result up to 64 bits. This instruction is irregular because it performs a 32-bit operation but produces a result which is not in a sign-extended 32-bit representation. The result is, in fact, a 64-bit representation and is always of a positive number. The instruction is useful because it provides a means to convert unsigned 32-bit integers up to 64-bit width in a single instruction.

Instruction	Summary
ADDZ.L source1,source2,result	add with zero-extend 32-bit

Table 31: Addition with zero-extend instruction

This instruction is commonly used in the form:

```
ADDZ.L Rm, R63, Rd ; 32-bit zero-extend of Rm into Rd
```

However, in cases where there is an unsigned addition at 32-bit precision followed by a conversion up to a 64-bit width, then both the addition and the conversion can be achieved using this one instruction.

Two multiply instructions are provided. MULS.L performs a multiplication of two signed 32-bit register values to give a 64-bit result. MULU.L multiplies two unsigned 32-bit register values and also gives a 64-bit result.Both MULS.L and MULU.L ignore the upper 32 bits of their register sources to give free narrowing type conversion on input.

Instruction	Summary
MULS.L source1,source2,result	multiply full 32-bit x 32-bit to 64-bit signed
MULU.L source1,source2,result	multiply full 32-bit x 32-bit to 64-bit unsigned

Table 32: Multiply instructions

A code sequence to perform a 32-bit by 32-bit multiply to give a 32-bit result is:

```
MULU.L Rm, Rn, Rd ; multiply Rm and Rn to 64-bit result
ADD.L Rd, R63, Rd ; convert down to a 32-bit result stored in Rd
```

This sequence can be used for both signed and unsigned 32-bit multiplies since the lower bits of a multiply are not affected by sign, and since the output representations of signed 32-bit and unsigned 32-bit numbers are the same. For the same reasons, the corresponding sequence with MULS.L gives an identical end result. In some cases, it is possible to eliminate the second instruction of this sequence by exploiting a free narrowing on subsequent instructions that read the multiply result.

A sequence for a full 64-bit by 64-bit multiply to give a 64-bit result is:

```
MULU.L Rm, Rn, Rd ; product of Rm lower by Rn lower into Rd
SHLRI Rm, 32, R1 ; right shift Rm upper half into R1
SHLRI Rn, 32, R2 ; right shift Rn upper half into R2
MULU.L R1, Rn, R3 ; product of Rm upper by Rn lower into R3
MULU.L R2, Rm, R4 ; product of Rn upper by Rm lower into R4
ADD R3, R4, R5 ; add two lower/upper products together into R5
SHLLI R5, 32, R6 ; left shift sum of lower/upper products into R6
ADD Rd, R6, Rd ; calculate final 64-bit result into Rd
```

This sequence can be used for both signed and unsigned 64-bit multiplication.

The sequences described above make use of shifting and bitwise instructions which are defined later. Registers numbers R1 to R6 are used as temporaries. The selected register allocation is arbitrary; other allocations can be more appropriate in practice.

5.5 Comparison instructions

The comparison instructions allow two register values to be compared with each other. All comparison instructions compare all 64 bits of the two sources. The result of a comparison is a boolean value: 0 indicates that the comparison was false, and 1 that the comparison was true. The result is stored into a register destination.

A minimum number of compare instructions are provided. There are no compares against immediate values, though compares with zero can be achieved using R63. Other constant values must be loaded in advance if required. The supported compares are to test two registers for equality (CMPEQ), to test whether one register is greater than another register in a signed sense (CMPGT), and to test whether one register is greater than another register in an unsigned sense (CMPGTU).

Instruction	Summary
CMPEQ source1,source2,result	compare equal 64-bit
CMPGT source1,source2,result	compare greater than 64-bit signed
CMPGTU source1,source2,result	compare greater than 64-bit unsigned

Table 33: Compare instructions

Other compares can be synthesized using the following standard equivalences:

```
(i ≠ j) ≡ NOT (i=j)
(i < j) ≡ (j > i)
(i ≤ j) ≡ NOT (i > j)
(i ≥ j) ≡ NOT (j > i)
```

Thus by swapping the operand order and providing subsequent boolean negations, as required, all compares can be generated. The boolean negation can be achieved using XORI with an immediate value of 1 (see *Section 5.6: Bitwise instructions on page 84*). In some cases, it is possible to eliminate the explicit boolean negation by folding it into the subsequent use of the compare result.

The compares are defined to operate on 64-bit integers. CMPEQ can operate on signed or unsigned 64-bit integers, while CMPGT is for signed 64-bit integers and CMPGTU is for unsigned 64-bit integers.

For the compares to have the correct effect on 32-bit integers, the sign-extended 32-bit representation should be used. CMPEQ can operate on signed or unsigned 32-bit integers, while CMPGT is for signed 32-bit integers and CMPGTU is for unsigned 32-bit integers.

It is important to note that unsigned comparisons can be performed directly at 64-bit width on unsigned 32-bit numbers, even though those numbers are held in a 32-bit sign-extended representation. The effect of the sign-extension is to add (264 - 232) onto the value of each of the unsigned 32-bit numbers in the range [231, 232). This translation from the unsigned 32-bit number space, via sign-extension, into an unsigned 64-bit number space preserves the ordering of all of these unsigned numbers. This means that unsigned 64-bit compares give the correct results.

Compares often occur in programs to control conditional branches. Rather than use a compare instruction followed by a branch based on whether that compare result was true or false, it is possible to fold the effect of the compare directly into the branch instruction. This form is preferred because it contains one less instruction. Comparison instructions are typically only used when the compare result is needed as an arithmetic value.

5.6 Bitwise instructions

All possible bitwise combinations of two bits, A and B, can be formed using an AND, OR, XOR, or ANDC operation, followed by an optional NOT. A full set of truth tables follows:

A	B	0	A AND B	A ANDC B	A	B ANDC A	B	A XOR B	A OR B
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Table 34: Bitwise operations

A	B	1	NOT (A AND B)	NOT (A ANDC B)	NOTA	NOT (B ANDC A)	NOT B	NOT (A XOR B)	NOT (A OR B)
0	0	1	1	1	1	1	1	1	1
0	1	1	1	1	1	0	0	0	0
1	0	1	1	0	0	1	1	0	0
1	1	1	0	1	0	1	0	1	0

Table 35: Bitwise operations (continued)

The instruction set supports AND, OR, XOR and ANDC operations directly.

Instruction	Summary
AND source1,source2,result	bitwise AND 64-bit
OR source1,source2,result	bitwise OR 64-bit
XOR source1,source2,result	bitwise XOR 64-bit
ANDC source1,source2,result	bitwise ANDC 64-bit

Table 36: Bitwise instructions

These instructions operate on two 64-bit sources to give a 64-bit result. The appropriate bitwise operation is performed independently on each of the 64 bit positions in the sources and result.

The instructions can also be used directly on 32-bit data. If the two 32-bit sources use a 32-bit sign-extended representations, then the output 64-bit result will also be in a 32-bit sign-extended representation. This happens because the same bitwise operation will be applied to the sign bit as to all of the upper 32 bits. It should be noted that these operations do not give a free narrowing on input for 32-bit integer values.

Bitwise instructions often occur with one operand as a constant. The instruction set therefore provides immediate forms of the supported bitwise operations.

Instruction	Summary
ANDI source1,source2,result	bitwise AND immediate 64-bit

Table 37: Bitwise with immediate instructions

Instruction	Summary
ORI source1,source2,result	bitwise OR immediate 64-bit
XORI source1,source2,result	bitwise XOR immediate 64-bit

Table 37: Bitwise with immediate instructions

Each of these instructions uses a sign-extended immediate, Note that an *andc* operation with an immediate is not provided since the complement can be folded into the value of the sign-extended immediate.

The NOT operation is supported using an XORI with -1:

```
XORI Rm, -1, Rd ; to bit-not Rm into Rd
```

For some bitwise operations, it is necessary to follow one of the provided bitwise instructions by a bitwise NOT. However, the most common operations are available in a single instruction.

5.7 Shift instructions

Three different styles of shifting are supported:

- Shift logical left (‘SHLL’): the value is shifted left by the specified amount and zero bits are inserted at the least significant end of the shift. In fact, the same shift left operation can be used for both logical and arithmetic operations since these operations are identical.
- Shift logical right (‘SHLR’): the value is shifted right by the specified amount and zero bits are inserted at the most significant end of the shift.
- Shift arithmetic right (‘SHAR’): the value is shifted right by the specified amount and sign bits are inserted at the most significant end of the shift, so that the sign of the result is the same as that of the source.

The above list gives the base mnemonic for the shift names. The shift amount can be specified by an immediate value (an immediate shift) or by a register value (a dynamic shift). The immediate shift instructions append an ‘I’ to the base shift mnemonic, while the dynamic shifts append a ‘D’. Shifting operations are supported at 32-bit and 64-bit width. The 32-bit shift is distinguished from the 64-bit shift by appending a trailing ‘.L’ to the shift mnemonic.

All of these options are orthogonal. The total number of shift operations supported is, therefore, three times two times two which is twelve.

For 64-bit shift instructions, only the low 6 bits of the shift amount are used. This means that the shift amount is modulo 64; for example, a shift of 64 has the same effect as a shift of 0.

The immediate 64-bit shifts are:

Instruction	Summary
SHLLI source1,source2,result	shift logical left immediate 64-bit
SHLRI source1,source2,result	shift logical right immediate 64-bit
SHARI source1,source2,result	shift arithmetic right immediate 64-bit

Table 38: Immediate 64-bit shift instructions

The dynamic 64-bit shifts are:

Instruction	Summary
SHLLD source1,source2,result	shift logical left dynamic 64-bit
SHLRD source1,source2,result	shift logical right dynamic 64-bit
SHARD source1,source2,result	shift arithmetic right dynamic 64-bit

Table 39: Dynamic 64-bit shift instructions

For 32-bit shift instructions, only the low 5 bits of the shift amount are used. This means that the shift amount is modulo 32; for example, a shift of 32 has the same effect as a shift of 0. The 32-bit shift instructions ignore the upper 32 bits of their sources, and write a 32-bit result which is sign-extended up to 64 bits. This arrangement means that they give a free narrowing type conversion on input, and always produce an output in a sign-extended 32-bit representation.

The immediate 32-bit shifts are:

Instruction	Summary
SHLLI.L source1,source2,result	shift logical left immediate 32-bit
SHLRI.L source1,source2,result	shift logical right immediate 32-bit
SHARI.L source1,source2,result	shift arithmetic right immediate 32-bit

Table 40: Immediate 32-bit shift instructions

The dynamic 32-bit shifts are:

Instruction	Summary
SHLLD.L source1,source2,result	shift logical left dynamic 32-bit
SHLRD.L source1,source2,result	shift logical right dynamic 32-bit
SHARD.L source1,source2,result	shift arithmetic right dynamic 32-bit

Table 41: Dynamic 64-bit shift instructions

Shifts are often used for bit-field manipulation and for scaling. They can be used as an efficient alternative for multiplication and division by 2^n. The value of n must not be negative and it must be less than the width of the shift instruction.

A logical left shift can be used for the signed or unsigned multiplication of a value by 2^n. A logical right shift can be used for the unsigned division of a value by 2^n. An arithmetic right shift can be used for the signed division of a value by 2^n. In this last case, however, the arithmetic right shift will give a division that rounds the result towards minus infinity. If a division is required that rounds the result towards zero, then the arithmetic right shift will give unexpected results for some negative input values. It is possible to correct the behavior of the arithmetic right shift with a short instruction sequence.

5.8 Miscellaneous instructions

The instruction set provides 5 integer instructions which do not fall naturally into any of the other categories.

Instruction	Summary
BYTEREV source,result	byte reversal
CMVEQ source1,source2,source3_result	conditional move if equal to zero
CMVNE source1,source2,source3_result	conditional move if not equal to zero
NOP	no operation
NSB source,result	count number of sign bits

Table 42: Miscellaneous instructions

The BYTEREV instruction reverses all 8 bytes in the source register, and stores the result in the destination register. Byte number i, where i is in [0, 7], is moved to byte number (7-i). This is particularly useful for converting data between little endian and big endian representations. If a byte reversal on a narrower data type is required, then a right shift can be used after the BYTEREV. For example, an 8-byte reversal requires a single instruction:

```
BYTEREV Rm, Rn ; to byte reverse an 8-byte integer
```

A 4-byte reversal requires two instructions:

```
BYTEREV Rm, Rn
SHARI Rn, 32, Rn ; to byte reverse a 4-byte integer
```

The conditional move instructions can be used to eliminate some conditional branches. For example, consider a basic block that is executed only if some condition is true, which computes some side-effect free expression and assigns it to some variable. Usually this is compiled into a conditional branch to guard execution of the basic block. With conditional moves, the branch can be eliminated and replaced by a conditional assignment of the expression result to the variable. This technique is generally called ‘if-conversion’. It allows small basic blocks to be merged into surrounding ones, and can remove unpredictable and costly branches.

The CMVEQ instruction moves source2 to result only if source1 is zero. The CMVNE instruction moves source2 to result only if source1 is not zero. Two conditional moves are provided in order to give a free negation through choice of the appropriate conditional move instruction.

Common uses of conditional move are evaluation of the maximum of two values:

```
CMPGT Rm, Rn, R0
CMVNE R0, Rm, Rn ; Rn = (Rm > Rn) ? Rm : Rn;
```

and the minimum of two values:

```
CMPGT Rm, Rn, R0
CMVEQ R0, Rm, Rn ; Rn = (Rm <= Rn) ? Rm : Rn;
```

The NOP instruction performs no operation. It can be used, for example, to pad a sequence of instructions up to a particular alignment boundary.

The NSB instruction counts the number of sign bits in its source register, subtracts 1 and stores the result in its destination register. The number of sign bits is the number of consecutive bits, including the most significant bit and moving down towards the least significant bit, that have the same bit value.

5.9 General-purpose register move

There is no dedicated instruction for moving one general-purpose register value into another. It is recommended that ORI is used with an immediate value of 0:

```
ORI Rm, 0, Rd ; move Rm into Rd
```

6 SHmedia memory instructions

6.1 Overview

Memory is byte addressed. The memory instructions provide access to data using little endian or big endian representations. Endianness is specified at power-on reset, and does not change thereafter. The mechanism for selecting between little endian and big endian operation is external to the CPU, and is not specified by the CPU architecture.

This chapter defines the general-purpose load and store instructions. Load and store instructions for floating-point registers are described separately in *Chapter 8: SHmedia floating-point on page 135*.

Load and store instructions transfer data between a register and memory. Some load instructions have signed and unsigned variants to perform the correct extension into the register. For byte (8-bit) and word (16-bit) objects, both signed and unsigned loads exist. For long-word (32-bit) objects, only signed loads are provided because all 32-bit objects are held in a sign-extended form in registers regardless of sign. For quad-word (64-bit) objects, there is no distinction between signed and unsigned.

Two different sets of load and store instructions are provided:

- The first set are defined in *Section 6.2: Aligned load and store instructions on page 92* and support naturally aligned data. This is where the address of the data is an exact multiple of the width of the access. If one of these instructions attempts a misaligned access, it will cause a misalignment exception.
- The second set are defined in *Section 6.3: Misaligned access support on page 94*. These instructions are used in short sequences to synthesize accesses to misaligned data without an exception. These instructions should be used where software cannot statically determine that the data is naturally aligned.

This chapter also describes instructions for synchronization and cache control.

6.2 Aligned load and store instructions

The provided instructions are summarized in *Table 43*.

Access	Mode	Signed byte (8 bits)	Unsigned byte (8 bits)	Signed word (16 bits)	Unsigned word (16 bits)	Long word (32 bits)	Quad-word (64 bits)
Load	indexed	LDX.B	LDX.UB	LDX.W	LDX.UW	LDX.L	LDX.Q
	displacement	LD.B	LD.UB	LD.W	LD.UW	LD.L	LD.Q
Store	indexed	STX.B		STX.W		STX.L	STX.Q
	displacement	ST.B		ST.W		ST.L	ST.Q

Table 43: Aligned load and store instructions

If the destination register of an aligned load instruction is R_{63}, then this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. In exceptional cases, no exception is raised and the prefetch has no effect. Further information on prefetch is given in *Section 6.6.1: Prefetch on page 102*.

Displacement addressing

For displacement addressing, the effective address is calculated by adding a displacement to a base pointer. The displacement is a sign-extended 10-bit immediate value and the base pointer is held in a general-purpose register. The immediate value is scaled by the size of the object accessed.

Instruction	Summary	Displacement scaling factor
LD.B base,offset,result	load 8-bit signed	1
LD.UB base,offset,result	load 8-bit unsigned	1
LD.W base,offset,result	load 16-bit signed	2
LD.UW base,offset,result	load 16-bit unsigned	2

Table 44: Aligned load instructions with displacement addressing

Instruction	Summary	Displacement scaling factor
LD.L base,offset,result	load 32-bit	4
LD.Q base,offset,result	load 64-bit	8

Table 44: Aligned load instructions with displacement addressing

Instruction	Summary	Displacement scaling factor
ST.B base,offset,value	store 8-bit	1
ST.W base,offset,value	store 16-bit	2
ST.L base,offset,value	store 32-bit	4
ST.Q base,offset,value	store 64-bit	8

Table 45: Aligned store instructions with displacement addressing

Indexed addressing

For indexed addressing, the effective address is calculated by adding a base pointer with an index. Both the base pointer and the index are held in general-purpose registers. Unlike displacement addressing, the index is not scaled by the size of the object accessed.

Instruction	Summary
LDX.B base,index,result	load indexed 8-bit signed
LDX.UB base,index,result	load indexed 8-bit unsigned
LDX.W base,index,result	load indexed 16-bit signed
LDX.UW base,index,result	load indexed 16-bit unsigned
LDX.L base,index,result	load indexed 32-bit
LDX.Q base,index,result	load indexed 64-bit

Table 46: Aligned load instructions with indexed addressing

Instruction	Summary
STX.B base,index,value	store indexed 8-bit
STX.W base,index,value	store indexed 16-bit
STX.L base,index,value	store indexed 32-bit
STX.Q base,index,value	store indexed 64-bit

Table 47: Aligned store instructions with indexed addressing

6.3 Misaligned access support

All the load and store instructions described so far throw a misalignment trap if used to access a data object that is not naturally aligned. Instructions are also included that can be used to construct efficient sequences for loading objects that are misaligned or with unknown alignment.

Support for loading and storing misaligned long-words and quad-words is provided. Separate instructions are used to access the low-part and high-part of misaligned data. Only displacement addressing is supported using a sign-extended 6-bit immediate. The immediate is not scaled to allow formation of misaligned addresses.

Instruction	Summary
LDHI.L base,offset,result	load misaligned high part 32-bit
LDLO.L base,offset,result	load misaligned low part 32-bit
LDHI.Q base,offset,result	load misaligned high part 64-bit
LDLO.Q base,offset,result	load misaligned low part 64-bit

Table 48: Misaligned load instructions

Instruction	Summary
STHI.L base,offset,value	store misaligned high part 32-bit
STLO.L base,offset,value	store misaligned low part 32-bit
STHI.Q base,offset,value	store misaligned high part 64-bit

Table 49: Misaligned store instructions

Instruction	Summary
STLO.Q base,offset,value	store misaligned low part 64-bit

Table 49: Misaligned store instructions

Misaligned load

The instructions described in this section can be used to load a misaligned long-word or quad-word object in 3 instructions. Instruction sequences for misaligned long-word loads return a sign-extended 32-bit result. The general form of a misaligned load sequence is as follows:

```
LDHI.L    Rptr, off+3, Rhi
LDLO.L    Rptr, off, Rlo
OR        Rhi, Rlo, Result
```

The address of the highest byte in the misaligned object is passed to the “load high part” instruction, while the address of the lowest byte in the misaligned object is passed to the “load low part” instruction. Usually, the immediate operand to the high part instruction is (n-1) more than the immediate operand to the low part instruction, where “n” is the object size in bytes.

Figure 26 shows a little endian example of loading a misaligned long-word.

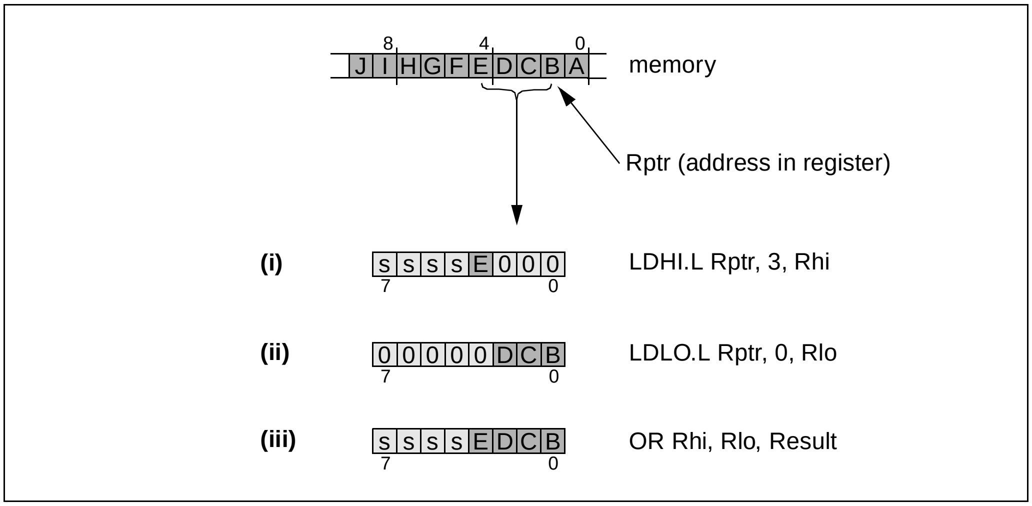


Figure 26: Misaligned load example

Support for loading misaligned words is not included, and alternative instruction sequences should be used. For example, the little endian load of an unsigned misaligned word (16 bits) can be achieved using:

```
LD.UB     Rbase, 0, Rtmp0 ; least significant byte
LD.UB     Rbase, 1, Rtmp1 ; most significant byte
MSHFLO.B  Rtmp0, Rtmp1, Result
```

The MSHFLO.B instruction (see *Section 7.18: Multimedia shuffles on page 130*) combines the two loaded bytes in the correct way to form an unsigned 16-bit word.

An example little endian sequence to load a signed misaligned word is:

```
LD.UB     Rbase, 0, Rtmp0 ; least significant byte
LD.B      Rbase, 1, Rtmp1 ; most significant byte
SHLLI     Rtmp1, 8, Rtmp1
OR        Rtmp0, Rtmp1, Result
```

Misaligned store

Storing a misaligned long-word or quad-word takes 2 instructions. The general form of a misaligned store sequence is as follows:

```
STHI.L    Rptr, off+3, Rvalue
STLO.L    Rptr, off, Rvalue
```

As for the misaligned load sequence, the address passed to the high part instruction should point to the highest byte of the misaligned object, while the address passed to the low part instruction should point to the lowest byte in the misaligned object. *Figure 27* illustrates a little endian example, storing a misaligned long-word.

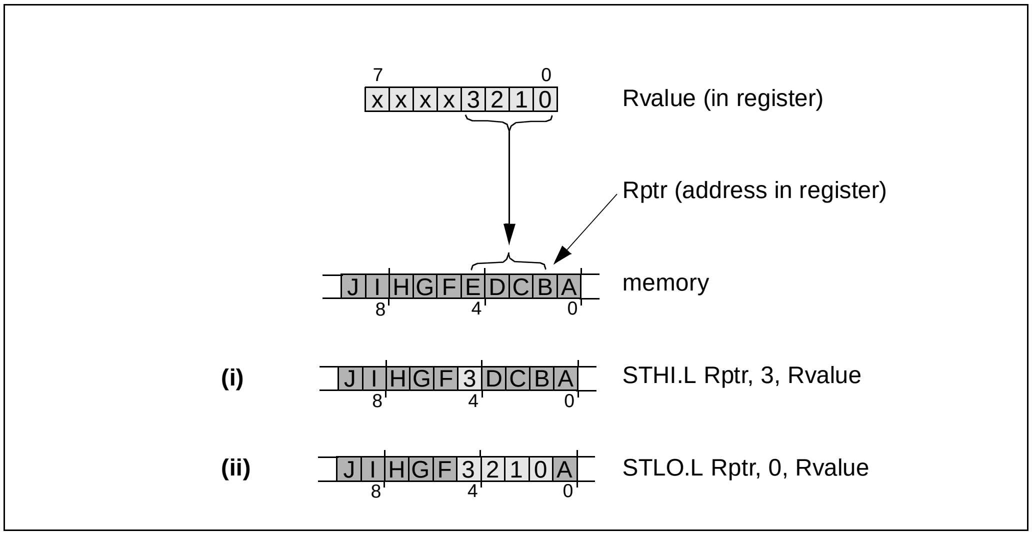


Figure 27: Misaligned store example

Support for storing misaligned words is not included, and alternative instruction sequences should be used. An example little endian sequence to store a misaligned word is:

```
ST.B     Rbase, 0, Rvalue ; least significant byte
SHLRI    Rvalue, 8, Rvalue
ST.B     Rbase, 1, Rvalue ; most significant byte
```

6.4 Memory properties

Data accesses issued by an instruction stream appear to execute in sequential program order as viewed from that instruction stream. However, these accesses do not necessarily appear to execute in that order as viewed by other observers. A data synchronization instruction is provided to allow ordering to be enforced upon data accesses (see *Section 6.5.3: Data synchronization on page 99*). This allows the instruction stream to guarantee that other observers view a particular access order.

A memory grain is a set of 8 contiguous bytes in memory whose base address is aligned to an 8-byte boundary. Each data access made by the instruction stream is fully contained within a grain of memory. This property is true for all load and store instructions, even for those that support misaligned access.

The behavior of concurrent data accesses to a grain from multiple memory users obeys an *atomicity* property. The memory system performs each data access on a grain atomically with respect to other data accesses on that grain. This means that the behavior of a set of accesses to a particular grain is identical to some completely sequentialized ordering of those accesses. For example, it is not possible to observe a grain of memory in a state where it has been partially updated by a write access.

The presence of memory management and caches has a significant effect on the properties of memory accesses. These effects are described in *Chapter 17: Memory management on page 271* and *Chapter 18: Caches on page 297*.

6.5 Synchronization

Instructions are provided for synchronization.

Instruction	Summary
SWAP.Q base,index,result_value	atomic swap in memory 64-bit
SYNCI	synchronize instructions
SYNCO	synchronize operand data

Table 50: Synchronization instructions

6.5.1 Atomic swap

The SWAP.Q instruction is an atomic read-modify-write operation on a memory location. SWAP.Q is typically used by software to synchronize multiple memory users through the memory system. It writes a new value into an 8-byte memory object and returns its previous contents. The memory system guarantees that the read and write parts of the swap instruction are implemented atomically on the target memory location with respect to any other accesses to that location.

Swap accesses are performed in memory not in the cache. This provides safe synchronization in the memory system regardless of the cache behavior.

If the MMU is disabled, then the cache state is bypassed and frozen with respect to data accesses including swap accesses. If the MMU is enabled, the actions performed by a SWAP.Q instruction for the various cache behaviors are:

- For device or uncached cache behavior, the effective address will not be cached. The swap is performed atomically in memory.

- For write-through cache behavior, the effective address can be cached but it will not be dirty. If it is cached, the cache block will be invalidated. The swap is performed atomically in memory.
- For write-back cache behavior, the effective address can be cached and can be dirty. If it is cached, the cache block will be purged (written-back if dirty, then invalidated). The swap is performed atomically in memory.

In each case, after the execution of the SWAP.Q instruction the targeted memory location will not be cached. Further information on caches, including cache terminology, can be found in *Chapter 18: Caches on page 297*.

6.5.2 Instruction synchronization

The SYNCI instruction is used to synchronize the instruction stream. Execution of a SYNCI ensures that all previous instructions are completed before any subsequent instruction is fetched. However, the SYNCI instruction does not ensure that the effects of those previous instructions on data memory have completed. Data synchronization can be achieved separately using the SYNCO instruction.

The SYNCI instruction does not cohere the state of any instruction cache. This must be achieved by explicit cache coherency instructions where required. This is described in *Chapter 18: Caches on page 297*.

SYNCI is used by software to:

- Synchronize instruction fetch after code has been loaded or modified (for example, see *Section 6.7.1: Synchronizing fetch with data writes on page 107*).
- Synchronize instruction fetch after instruction translations have been modified (also see *Section 16.8: Instruction synchronization on page 235*).
- Stop speculative execution of subsequent instructions (for example, see *Section 18.11.2: Speculative memory access when MMU is disabled on page 312*).

6.5.3 Data synchronization

The SYNCO instruction is used to synchronize data operations. Data operations include load, store, swap, prefetch, allocate and data cache coherency instructions. The SYNCO instruction imposes an ordering on data operations that is visible to other memory users.

Execution of a SYNCO ensures that all data operations from previous instructions are completed before any data access from subsequent instructions are started.

The SYNCO instruction itself does not complete until all data operations from previous instruction have completed. A sequence of a SYNCO instruction followed by a SYNCI instruction guarantees that all previous instructions, *and all previous data operations*, are completed before any subsequent instruction is fetched.

The SYNCO instruction does not cohere the state of any data cache. This must be achieved by explicit cache coherency instructions where required. This is described in *Chapter 18: Caches on page 297*.

SYNCO is used by software to:

- Order accesses to a memory location that is shared with another memory user.
- Order accesses to a device memory location.
- Flush any write buffering.
- Prevent memory accesses being merged or deleted.
- Order cache coherency instructions with respect to memory accesses.
- Order configuration register instructions with respect to memory accesses.

6.5.4 Implementation aspects

An implementation can provide mechanisms to optimize instruction fetch. These mechanisms could include, but are not limited to, the following:

- Instruction prefetching: this is a technique to reduce instruction fetch latency by fetching instructions before they are needed.
- Instruction buffering: this is a technique to reduce instruction fetch latency by holding instructions in a buffer close to the CPU, perhaps associated with the target registers.

SYNCI will cause the implementation to invalidate any such state to ensure that subsequent instructions are refetched.

An implementation can provide mechanisms to optimize data access. These mechanisms could include, but are not limited to, the following:

- Write buffering: this is a technique where written data is held in a buffer before been flushed out to memory at some later point. Write buffers can enhance memory performance by deferring and gathering writes.

  SYNCO will cause the implementation to flush any such state to memory. This ensures that the previous accesses propagate through to memory.

- Access reordering: this is a technique where a sequence of accesses to memory locations are reordered by the implementation. An implementation must maintain sufficient ordering to honor data dependencies through memory dependencies as observed from the CPU, but can otherwise reorder accesses arbitrarily.

  SYNCO imposes an ordering on accesses generated by the CPU. Execution of a SYNCO ensures that all data operations from previous instructions are completed before any data access from subsequent instructions are started.

6.6 Cache instructions

There are 3 categories of cache instruction: prefetch, allocate and coherency instructions. These instructions allow software to control and optimize cache operation in a largely implementation-independent manner.

Further information on caches can be found in *Chapter 18: Caches on page 297*.

The available instructions are summarized below.

Instruction	Summary
ALLOCO base,offset	allocate operand cache block
ICBI base,offset	instruction cache block invalidate
OCBI base,offset	operand cache block invalidate
OCBP base,offset	operand cache block purge
OCBWB base,offset	operand cache block write-back
PREFI base,offset	prefetch instruction cache block

Table 51: Cache instructions

These instructions compute an effective address by adding their base and offset operands. The base operand is held in a general-purpose register. The offset operand is a 6-bit immediate value which is then scaled by 32 and sign-extended. Thus, the offset has a value 32i where i is in the range [-32, 31). Note that the scaling factor is fixed at 32 regardless of the cache block size of the implementation.

There is no misalignment check on these instructions. The calculated effective address is automatically aligned downwards to the nearest exact multiple of the cache block size.

Most of the cache instructions have no functional effect on the semantics of the memory model when viewed solely from the instruction stream. However, ALLOCO and OCBI can result in observable effects on the memory model. These instructions can modify the value of memory locations, and the number of modified locations is determined by the cache block size. This value is implementation specific, and special care should therefore be exercised when using these instructions if portability to implementations with a difference cache block size is desired.

6.6.1 Prefetch

The architecture provides two mechanisms for software-directed prefetching from a specified effective address:

- The PREFI instruction is an instruction prefetch.
- An aligned load instruction, where the destination is R_{63}, is a data prefetch.

Instruction prefetch behaves much like an instruction fetch, except that it is software-directed. Data prefetch behaves much like a read access, except that data is loaded into a cache block rather than a register. Prefetches behave like normal accesses with respect to cache behavior and cache paradoxes.

There is no misalignment check for prefetches. The provided effective address is automatically aligned downwards to the nearest exact multiple of the cache block size. This applies to both instruction prefetches and data prefetches.

The generic architecture states that a prefetch is a performance hint to the implementation. A prefetch informs the implementation that there is likely to be a performance benefit in arranging for that data to be prefetched into the cache. Prefetches affect timing but not semantics.

It is implementation-specific as to whether a prefetch will be performed. For example, an implementation could choose to ignore all prefetches, or an implementation could choose to ignore a particular prefetch depending on the prevailing conditions.

There are a number of scenarios where a prefetch has no effect:

- An implementation chooses to ignore the prefetch.
- A prefetch when the MMU is disabled has no effect.
- A prefetch with device or uncached behavior has no effect.
- If an implementation does not provide an instruction cache or a unified cache, then instruction prefetch has no effect. If an implementation does not provide an operand cache or a unified cache, then data prefetch has no effect.

- Prefetches do not raise address error, translation miss or protection exceptions. If there is an address error, or a translation is not available, or a protection check fails, then the prefetch has no effect. These properties allow software to speculate prefetches. Note that prefetches are automatically aligned to the cache block size, and are not checked for misalignment.

6.6.2 Allocate

The architecture provides an instruction, ALLOCO, to allocate an operand cache block for a specified effective address. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size.

The allocate instruction provides a hint to the implementation that the allocated operand cache block need not be fetched from memory. It is implementation-specific as to whether the operand cache block will be fetched from memory or not.

ALLOCO is specifically designed to be used in combination with write-back cache behavior. Typically, ALLOCO is used to allocate an operand cache line which is then completely over-written with new data using store instructions, and subsequently written-back. ALLOCO can eliminate an unnecessary cache block fetch from memory, avoiding memory read latency and reducing memory bandwidth.

ALLOCO is checked for address error, translation miss and protection exceptions just like a data write to that address. There is no misalignment check. ALLOCO behaves like a normal access with respect to cache behavior and cache paradoxes.

In some situations an allocate instruction has no effect (apart from the detection of exception cases):

- An allocate when the MMU is disabled has no effect.
- An allocate with device or uncached behavior has no effect.
- If an implementation provides neither an operand cache nor a unified cache, then allocate has no effect.

In all other cases, the value of each location in the memory block targeted by an ALLOCO becomes architecturally undefined. However, ALLOCO will not reveal any data which would break the privilege and protection models. An example implementation of ALLOCO could have the following properties:

- In user mode, the values of the targeted memory locations seen by that user mode thread could be unchanged, filled with some implementation-defined pattern or filled with some other data that is also visible to that thread.

- In privileged mode, the targeted memory locations could contain any value. This is allowed since privileged threads can arrange visibility of any memory state.

Other implementations of ALLOCO are possible with different effects on the targeted memory locations. Software must not rely on these values, otherwise compatibility will be impaired.

6.6.3 Cache coherency

The architecture provides a set of cache coherency instructions that allow the cache to be managed by software. These instructions are:

- ICBI: to invalidate an instruction cache block.
- OCBI: to invalidate an operand cache block.
- OCBP: to purge an operand cache block.
- OCBWB: to write-back an operand cache block.

For an invalidation, the cache block is discarded without any write-back to memory. For a purge, the cache block is written back to memory if dirty, and then discarded. For a write-back, the cache block is written back memory if dirty, but not discarded. Write-back is also known as flush.

These instructions operate directly on the state of the cache. In some respects, these instructions behave quite differently to normal memory accesses:

- The OCBI, OCBP and OCBWB instructions update the state of the cache even if the MMU is disabled. In this case, the effective address calculated by the cache coherency instruction is mapped to a physical address using an identity translation. The MMU is not consulted since it is disabled, and the cache is accessed using a look-up by physical address. TLB misses and protection violations cannot occur when the MMU is disabled.
- It is implementation dependent as to whether ICBI updates the state of the instruction cache when the MMU is disabled. If the instruction cache supports look-up by physical address, the effective address calculated by ICBI is mapped to a physical address using an identity translation. The MMU is not consulted since it is disabled, and the cache is accessed using a look-up by physical address. However, if the instruction cache does not support look-up by physical address, then the ICBI has no effect when the MMU is disabled. In either case, TLB misses and protection violations cannot occur when the MMU is disabled.
- These instructions update the state of the cache regardless of the programmed cache behavior.

- These instructions are not susceptible to cache paradoxes.

In general, these instructions have a guaranteed effect on the cache regardless of the cache and MMU configuration.

There are no restrictions placed on execution of cache coherency instructions directly out of the instruction cache. When the MMU is enabled these instructions can be executed from an instruction translation with a cachable page, and therefore these instructions (including ICBI) can be fetched from the instruction cache. The synchronization requirements are described separately in *Synchronization of cache coherency instructions on page 106*.

ICBI and OCBI have the same effect on a unified cache implementation, though note that their exception checks are different. In this case, software should still ensure that ICBI is used for instruction invalidation, and OCBI for data invalidation. This enhances portability to implementations with split caches.

Physical and effective coherency

OCBI, OCBP and OCBWB perform cache coherency on physical memory. These instructions use an effective address to identify locations in physical memory which are to be cohered. The achieved coherency applies to all aliases of that physical memory in the effective address space.

However, ICBI is only guaranteed to achieve coherency on effective memory. This instruction uses an effective address to identify locations in effective memory which are to be cohered. The achieved coherency applies only to the effective address and effective address space seen by that ICBI. It does not necessarily apply to other mappings of that location with different effective addresses or in a different effective address space.

An implementation can choose to provide stronger coherency than this (for example, by implementing this as coherency on physical memory), but software must not rely on this behavior where portability is required.

Exception checking

These instructions are checked for address error, translation miss and protection exceptions in a similar way to memory accesses. There is no misalignment check.

OCBI is checked like a data write to that address. It is considered to be a write because its execution can cause memory values to change (as viewed from the instruction stream).

OCBP and OCBWB are checked for readability or writability to that address. Thus, a protection exception is raised if both reads and writes are prohibited. This is a read exception because the execution of these instructions does not cause memory values to change (as viewed from the instruction stream). These instructions propagate previously written data beyond the cache, and are not considered to be writes.

ICBI is checked like an instruction fetch from that address. ICBI checks for address error and raises an IADDERR exception if this check fails. The implementation then determines whether there is an entry in the instruction cache for this ICBI to invalidate. Some implementations perform a translation look-up to determine this; these implementations will raise an ITLBMISS exception if there is no translation available. Other implementations can determine this without raising ITLBMISS.

Thus, whether an ITLBMISS exception can be raised by ICBI is implementation dependent. If there is no entry in the instruction cache for this ICBI to invalidate, then the ICBI can complete as no invalidation is required. If there is an entry in the instruction cache for this ICBI to invalidate, then a check is made for protection violation prior to invalidation. If a protection violation occurs, the instruction executes to completion without exception launch, but does not affect the state of the instruction cache. This property does not weaken the protection model, but it does make it harder to detect some software bugs in ICBI code sequences. This behavior is specified to allow simpler ICBI implementation.

If an implementation does not provide an instruction cache or a unified cache, then ICBI is checked for exceptions but otherwise behaves as a no-op. If an implementation does not provide an operand cache or a unified cache, then OCBI, OCBP and OCBWB are checked for exceptions but otherwise behave as no-ops.

Synchronization of cache coherency instructions

Explicit synchronization instructions are required to synchronize the effects of cache coherency instructions:

- SYNCI must be used to guarantee that previous ICBI instructions have completed their invalidation on the instruction cache.
- SYNCO must be used to guarantee that previous OCBI, OCBP and OCBWB instructions have completed their operation on the operand cache and on memory.

Typically, one SYNCI or SYNCO instruction is used to synchronize a whole series of previous cache coherency instructions.

6.7 Example code sequences

This section describes example code sequences that use the synchronization and cache instructions.

6.7.1 Synchronizing fetch with data writes

Software can use stores to write a sequence of instructions into memory. For example, this could be to load a program into memory, to modify code, to set software break-points or for just-in-time compilation. The recommended procedure to synchronize instruction fetch with respect to a set of data writes is:

1. Execute the stores.
2. If the stores were to memory using write-back cache semantics, then flush all data cache blocks in the stored address range.
3. Execute a SYNCO instruction to ensure that the memory accesses in steps *1* and *2* have completed.
4. If the instruction addresses are cachable, then invalidate all instruction cache blocks in the instruction address range.
5. Execute a SYNCI instruction to ensure that all previous steps have completed and to synchronize instruction fetch.
6. Code can now be executed from the synchronized instruction address range.

7 SHmedia multimedia instructions

7.1 Overview

Although this is a 64-bit architecture, many data elements are much smaller. The architecture defines a set of multimedia operations which perform an operation on several small elements concurrently.

An extensive set of data parallel arithmetic and data manipulation operations are provided. These include conversions, addition, subtraction, absolute value, sum of absolute differences, shifts, comparisons, multiplies, multiply accumulate, shuffle, conditional move, permute and extract.

Data formats include packed 8-bit, 16-bit and 32-bit integers. Support is provided for signed and unsigned integers, and signed fractional formats with no integral bits. Not all formats are supported at all element sizes.

Multimedia instruction mnemonics are constructed with the prefix ‘M’. For instance the multimedia 32-bit add is called **MADD.L**.

7.2 Multimedia formats

Table 52 shows the supported multimedia formats.

8 x 8-bit multimedia data

8-bit data	8-bit data	8-bit data	8-bit data	8-bit data	8-bit data	8-bit data	8-bit data
Element 7	Element 6	Element 5	Element 4	Element 3	Element 2	Element 1	Element 0
63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0

4 x 16-bit multimedia data

16-bit data	16-bit data	16-bit data	16-bit data
Element 3	Element 2	Element 1	Element 0
63 48	47 32	31 16	15 0

2 x 32-bit multimedia data

32-bit data	32-bit data
Element 1	Element 0
63 32	31 0

Table 52: Multimedia data types

Table 53 shows the integral formats and ranges supported.

Element width	Element type	Field widths			Field locations			Range	Precision
		Sign	Int	Frac	Sign	Int	Frac		
8-bit integer	Signed byte (B)	1	7	-	7	6-0	-	[-128, +127]	1
	Unsigned byte (UB)	-	8	-	-	7-0	-	[0, 255]	1
16-bit integer	Signed word (W)	1	15	-	15	14-0	-	[-32678, +32767]	1
	Unsigned word (UW)	-	16	-	-	15-0	-	[0, 65535]	1
32-bit integer	Signed long-word (L)	1	31	-	31	30-0	-	[-2147483648, +2147483647]	1

Table 53: Integral representation

The architecture additionally supports some formats with fractional bit significance. These formats are useful in signal processing and graphical algorithms. The fractional formats are held in 16-bit or 32-bit fields, but the significance of each bit differs from the integer weight. The fractional format has a single sign bit and the remainder are fractional weight bits. There are no integral weight bits. The most positive and most negative representable values have the same bit pattern as the same-width signed integral formats.

Table 54 shows the formats of the 16-bit and 32-bit fractional data types.

Element width	Element type	Field widths			Field locations			Range	Precision
		Sign	Int	Frac	Sign	Int	Frac		
16-bit fractional	Signed fractional word (W)	1	-	15	15	-	14-0	[-1, +1)	~31x10^{-6}
32-bit fractional	Signed fractional long-word (L)	1	-	31	31	-	30-0	[-1, +1)	~466x10^{-12}

Table 54: Fractional representation

The fractional formats require special multiply instructions, to extract the correct bits from the full (double sized) result. The same addition, subtraction and scaling instructions can be used on both the integral and fractional data formats.

If other formats with different placements of the binary point are to be multiplied, either the full double-length result must be explicitly scaled, or the source operands scaled so that one of the supported multiplications can be used.

Fractional data formats are described with two numbers separated by a '.'. These specify the number of bits before (including the sign bit) and after the binary point. 16-bit fractional numbers correspond to 1.15 and 32-bit fractional numbers correspond to 1.31.

Certain operations naturally produce a double-length result. As this cannot be returned directly, these operations are separated into two instructions, one for each half of the result. These are denoted by '**LO**' and '**HI**' suffices, for the low half and high half of the double-length result respectively.

7.2.1 Mathematics

Arithmetic on multimedia datatypes is defined as follows.

Signed numbers are held in 2's complement form.

Some operations behave in a modulo fashion. Operands producing an intermediate result that is outside the representable range, return the least significant *n* bits of the intermediate result where *n* is the element width (as defined in *Table 53*). For example, incrementing the most positive representable value of an unsigned type gives a result of zero.

Some operations behave in a saturating manner. Operands producing an intermediate result that is outside the representable range, return the most positive or most negative representable value as appropriate. This saturating behavior is emphasized in the instruction names by the suffix '**S**' being appended to the major operator identifier (for example, MADDS.W). Where there is no possibility of saturating, the 'S' suffix is omitted.

7.2.2 Rounding

Because of the fractional data formats, certain operations produce an intermediate result which has more fractional bits of precision than the result data type. The intermediate result is rounded to the representable type.

Two rounding modes are provided,

Round towards Minus (RM)	The result is the greatest representable number which is no greater than the precise result. This rounding mode, sometimes termed *truncation*, is provided by ignoring the excess precision bits. It guarantees that each representable value has equal weighting.
Round towards Nearest Positive (RP)	The result is the representable number closest to the precise result, except when the precise result lies exactly between two representable values. When this occurs the result is the greater representable value.

Most of the fractional multiplies provided employ round toward minus. Only one instruction is provided that uses round towards nearest positive: see *Section 7.12: Multimedia full-width multiplies on page 124*.

Figure 28 shows the behavior of these two rounding modes.

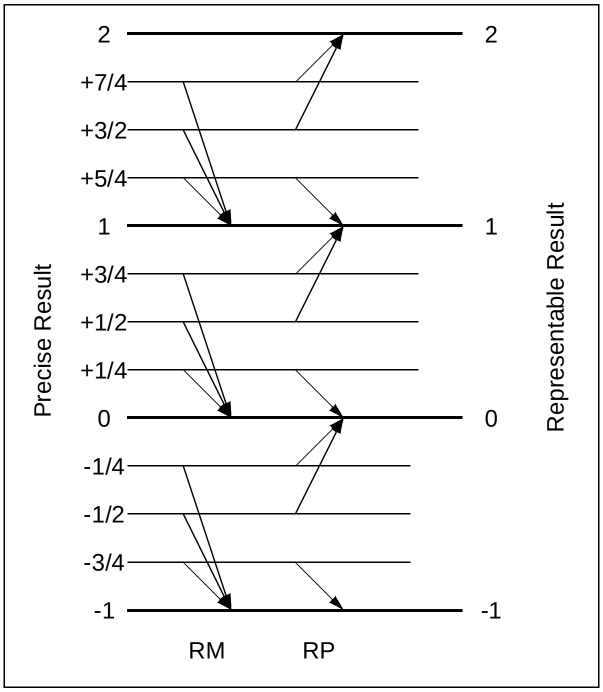


Figure 28: Rounding

7.2.3 MOSTPOS and MOSTNEG

This chapter uses the following notation:

MOSTPOS indicates the most positive representable value in a signed range.

MOSTNEG indicates the most negative representable value in a signed range.

For an integer signed format containing n bits, these are defined as:

MOSTPOS = $2^{n-1}-1$

MOSTNEG = $-(2^{n-1})$

For a fractional signed format containing n bits, these are defined as:

MOSTPOS = $1 - 2^{1-n}$

MOSTNEG = -1

7.3 Multimedia conversions

A conversion that reduces the size of packed elements is termed “down conversion”, while a conversion that increases the size of packed elements is termed “up conversion”. Down conversion can either be modulo or saturating, while up conversion can either be sign or zero extending.

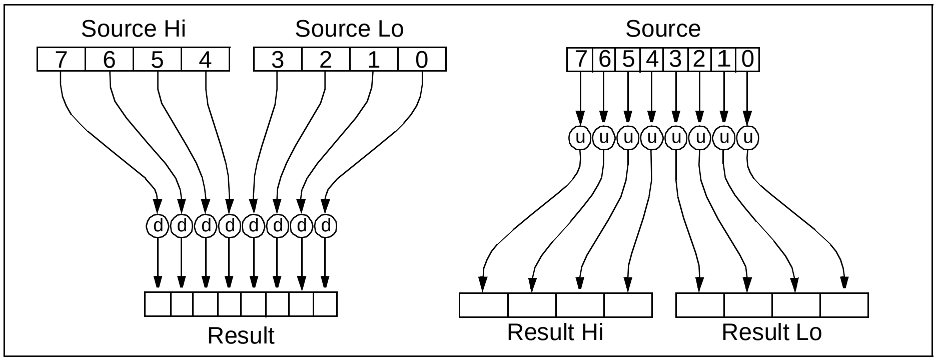


Figure 29: Example conversions

Only saturating down conversions are supported by single instruction sequences.

Instruction	Summary
MCNVS.WB source1,source2,result	Multimedia convert signed 16-bit to signed 8-bit after saturation
MCNVS.WUB source1,source2,result	Multimedia convert signed 16-bit to unsigned 8-bit after saturation
MCNVS.LW source1,source2,result	Multimedia convert signed 32-bit to signed 16-bit after saturation

Table 55: Multimedia conversion instructions

Table 56 shows the provided saturating down conversion instructions. Some other conversions can be performed using short instruction sequences. These employ additional multimedia instructions defined elsewhere in this chapter.

From→ / To↓	8-bit unsigned integer	8-bit signed integer	16-bit signed integer	32-bit signed integer
8-bit unsigned integer		MCMPGT.UB MCMV	MCNVS.WUB	MCNVS.LW MCNVS.WUB
8-bit signed integer	MCMPGT.UB MCMV		MCNVS.WB	MCNVS.LW MCNVS.WB
16-bit signed integer	MSHFLO.B MSHFHI.B	MCMPGT.UB MSHFLO.B MSHFHI.B		MCNVS.LW
32-bit signed integer	MSHFLO.B MSHFHI.B MSHFLO.W MSHFHI.W	MCMPGT.UB MSHFLO.B MSHFHI.B MCMPGT.W MSHFLO.W MSHFHI.W	MCMPGT.W MSHFLO.W MSHFHI.W	

Table 56: Multimedia conversions

7.4 Multimedia addition and subtraction

Multimedia addition and subtraction are provided.

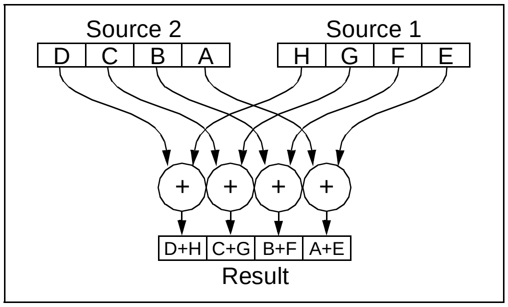


Figure 30: Multimedia addition

Both modulo (wrapping) and saturating forms are provided, although not all forms are provided at all sizes. On packed 8-bit elements the saturating operations are unsigned, while for packed 16-bit and packed 32-bit elements the saturating operations are signed.

Sign is unimportant in modulo arithmetic, and so the modulo add and subtract instructions can be used for both signed and unsigned types. Since bit significance is unimportant in addition and subtraction, the same set of instructions can be used to operate on both integral and fractional types.

Instruction	Summary
MADDS.UB source1,source2,result	Multimedia add unsigned 8-bit with saturation
MADD.W source1,source2,result	Multimedia add 16-bit
MADDS.W source1,source2,result	Multimedia add signed 16-bit with saturation
MADD.L source1,source2,result	Multimedia add 32-bit
MADDS.L source1,source2,result	Multimedia add signed 32-bit with saturation
MSUBS.UB source1,source2,result	Multimedia subtract unsigned 8-bit with saturation
MSUB.W source1,source2,result	Multimedia subtract 16-bit
MSUBS.W source1,source2,result	Multimedia subtract signed 16-bit with saturation
MSUB.L source1,source2,result	Multimedia subtract 32-bit
MSUBS.L source1,source2,result	Multimedia subtract signed 32-bit with saturation

Table 57: Multimedia addition and subtraction instructions

7.5 Multimedia absolute value

These instructions perform an absolute value operation on each signed element in a packed vector.

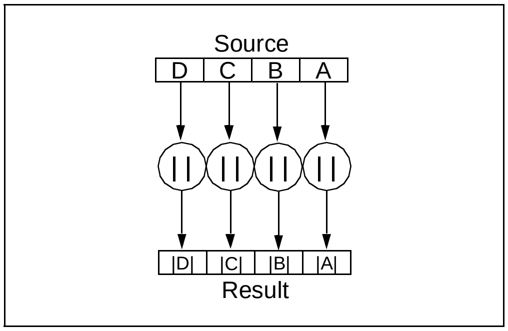


Figure 31: Multimedia absolute value

Taking the absolute value of the most negative representable value would produce an unsigned result out of the representable range. In this case the result is saturated to the most positive representable value.

Instructions to operate on packed 16-bit and packed 32-bit vectors are provided.

Instruction	Summary
MABS.W source,result	multimedia absolute value signed 16-bit with saturation
MABS.L source,result	multimedia absolute value signed 32-bit with saturation

Table 58: Multimedia absolute value

7.6 Multimedia sum of absolute differences

This instruction performs pair-wise absolute difference operations on two vectors of packed unsigned 8-bits, then sums the results and adds the sum to an accumulation. The operation being performed is:

$$\boxed{r = r + \sum_{i=0}^{7} |a_i - b_i|}$$

Instruction	Summary
MSAD.UBQ source1,source2,source3_result	Multimedia sum of absolute differences of unsigned 8-bit

Table 59: Multimedia sum of absolute differences

The MSAD.UBQ instruction consists of 24 elementary operations (8 subtractions, 8 absolute values and 8 additions) performed on byte-wide data.

7.7 Multimedia left shifts

These instructions perform a left shift operation on each packed element. Each element is shifted by the same amount. Only the least significant 4 or 5 bits of the shift amount are used, the remaining significant bits are ignored. This is consistent with the general-purpose shift instructions.

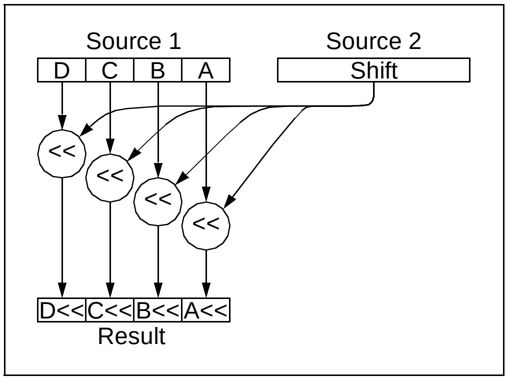


Figure 32: Multimedia 16-bit left shift

Both logical and saturating versions of the left shifts are provided. Sign is unimportant for a non-saturating left shift, and so the logical left shift instructions can be used for both signed and unsigned types. The saturating left shifts are provided only in signed form.

Left shifts are provided for packed 16-bit and packed 32-bit vectors.

Instruction	Summary
MSHLLD.W source,amount,result	Multimedia shift logical left dynamic 16-bit
MSHLLD.L source,amount,result	Multimedia shift logical left dynamic 32-bit
MSHALDS.W source,amount,result	Multimedia shift arithmetic left dynamic 16-bit with saturation
MSHALDS.L source,amount,result	Multimedia shift arithmetic left dynamic 32-bit with saturation

Table 60: Multimedia left shift instructions

7.8 Multimedia arithmetic right shifts

These instructions perform an arithmetic (signed) right shift operation on each packed element. Each element is shifted by the same amount. Only the least significant 4 or 5 bits of the shift amount are used, the remaining significant bits are ignored. This is consistent with the general-purpose shift instructions.

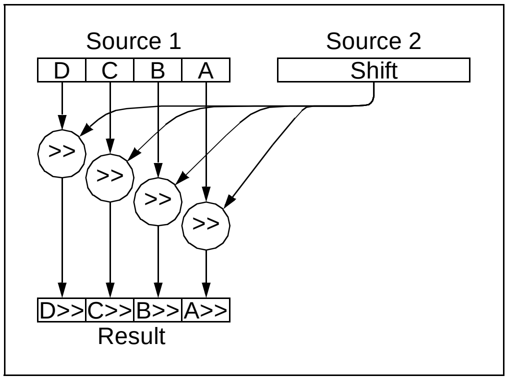


Figure 33: Multimedia 16-bit right shift

Arithmetic right shifts are provided for packed 16-bit and packed 32-bit vectors.

Instruction	Summary
MSHARD.W source,amount,result	multimedia shift arithmetic right dynamic 16-bit
MSHARD.L source,amount,result	multimedia shift arithmetic right dynamic 32-bit

Table 61: Multimedia arithmetic right shift instructions

7.9 Scalar arithmetic right shift with saturation

This instruction performs an arithmetic (signed) right shift on a signed, scalar 64-bit element, then saturate that single element into the 16-bit signed range $[-2^{15},+2^{15})$.

The result of this instruction is a 16-bit scalar integer value. The upper 48 bits of the result are sign extensions of bit 15. This operation is useful for reducing multiply-accumulate results.

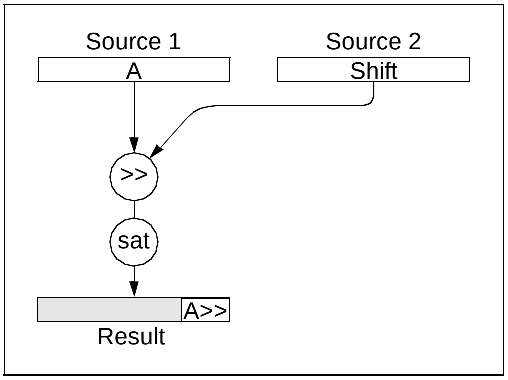


Figure 34: Scalar 64-bit Right shift with saturation to 16-bit

Instruction	Summary
MSHARDS.Q source,amount,result	multimedia shift arithmetic right dynamic with saturation to signed 16-bit

Table 62: Scalar arithmetic shift with saturation

7.10 Multimedia logical right shifts

These instructions perform a logical (unsigned) right shift operation on each packed element. Each element is shifted by the same amount. Only the least significant 4 or 5 bits of the shift amount are used, the remaining significant bits are ignored. This is consistent with the general-purpose shift instructions.

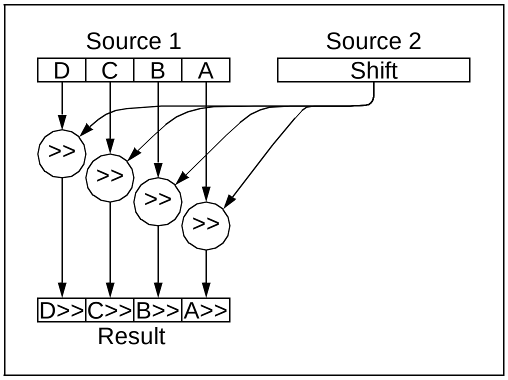


Figure 35: Multimedia 16-bit right shift

Logical right shifts are provided for packed 16-bit and packed 32-bit vectors.

Instruction	Summary
MSHLRD.W source,amount,result	Multimedia shift logical right dynamic 16-bit
MSHLRD.L source,amount,result	Multimedia shift logical right dynamic 32-bit

Table 63: Multimedia logical right shift instructions

7.11 Multimedia comparisons

Multimedia comparisons are provided that return all ones for true and all zeroes for false in each element within the packed result. This allows the results to be used directly as masks.

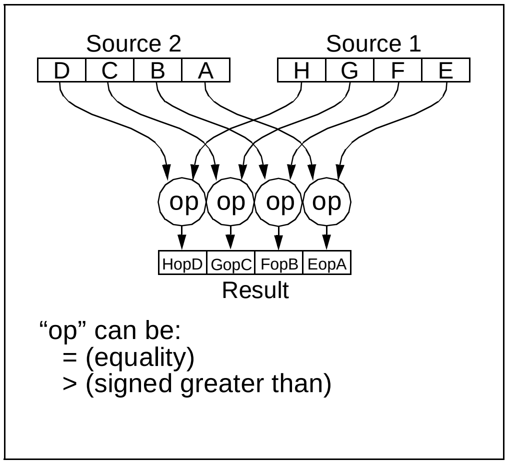


Figure 36: Multimedia comparisons

Sign is unimportant when comparing for equality, so only one flavor of compare for equality need be provided. For the greater-than compares, unsigned versions are provided for packed 8-bit elements, while signed versions are provided for packed 16-bit and packed 32-bit elements.

Instruction	Summary
MCMPEQ.B source1,source2,result	Multimedia compare equal 8-bit
MCMPEQ.W source1,source2,result	Multimedia compare equal 16-bit
MCMPEQ.L source1,source2,result	Multimedia compare equal 32-bit
MCMPGT.UB source1,source2,result	Multimedia compare greater than unsigned 8-bit
MCMPGT.W source1,source2,result	Multimedia compare greater than signed 16-bit
MCMPGT.L source1,source2,result	Multimedia compare greater than signed 32-bit

Table 64: Multimedia compare instructions

The sense of “equals” and “greater than” can be inverted to “not equals” and “less than or equal to” respectively by a bitwise inversion of the comparison results. Also, “less than” can be obtained by switching the source operands and using “greater than”. Finally, “greater than or equal to” can be obtained by switching the source operands, using “greater than” and then a bitwise inversion of the comparison result.

Since the comparison result often feeds into a packed conditional move (see *Section 7.21: Multimedia extract on page 133*) this inversion can often be for free, simply by switching the source operands to the conditional move.

7.12 Multimedia full-width multiplies

Multiplication of packed 16-bit quantities giving full-width results is provided.

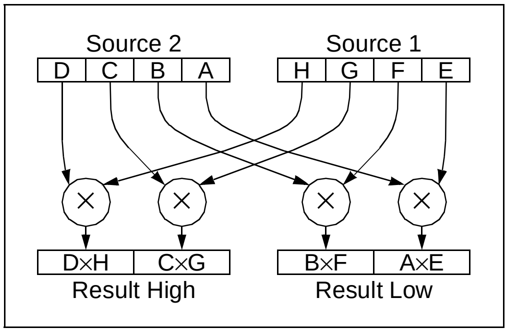


Figure 37: Multimedia 16-bit full multiplication

Since the resulting elements are twice the size of the inputs, two instructions are required; one to produce the low half of the results, and another to produce the high half.

Instruction	Summary
MMULLO.WL source1,source2,result	multimedia full multiply signed 16-bit low
MMULHI.WL source1,source2,result	multimedia full multiply signed 16-bit high

Table 65: Multimedia full-width multiplication instructions

7.13 Multimedia multiplies

Multiplications are provided for packed 16-bit and packed 32-bit types, in integral and fractional forms.

Integral multiplies use modulo arithmetic. For fractional multiplies where a multiplication of MOSTNEG by MOSTNEG occurs, the result, which would otherwise be outside the representable range, is saturated to MOSTPOS. The instruction provides rounding towards minus.

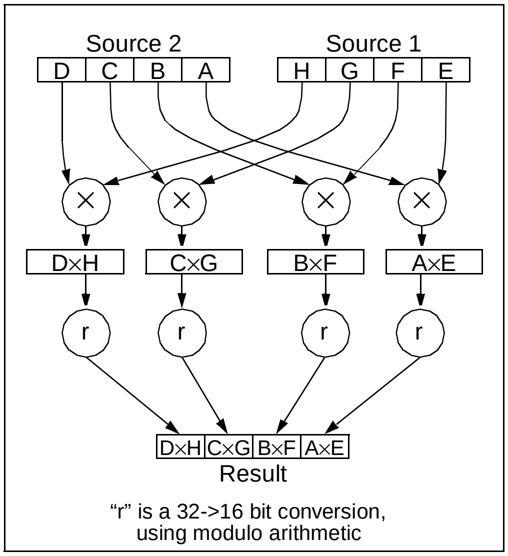


Figure 38: Multimedia 16-bit multiplies

Instruction	Summary
MMUL.W source1,source2,result	Multimedia multiply 16-bit
MMUL.L source1,source2,result	Multimedia multiply 32-bit
MMULFX.W source1,source2,result	Multimedia fractional multiply signed 16-bit
MMULFX.L source1,source2,result	Multimedia fractional multiply signed 32-bit

Table 66: Multimedia multiplication instructions

7.14 Multimedia multiply with rounding

A multiplication instruction is provided for packed 16-bit, fractional types that performs a rounding using the “round nearest positive” mode. The 32-bit intermediate results are rounded back to 16-bit fractional form by performing a “round nearest positive” rounding on the lower 16 fractional bits.

As with the other fractional multiplies, the special case of MOSTNEG by MOSTNEG is saturated to yield MOSTPOS.

Instruction	Summary
MMULFXRP.W source1,source2,result	Multimedia fractional multiply signed 16-bit, round nearest positive

Table 67: Multimedia multiply with rounding

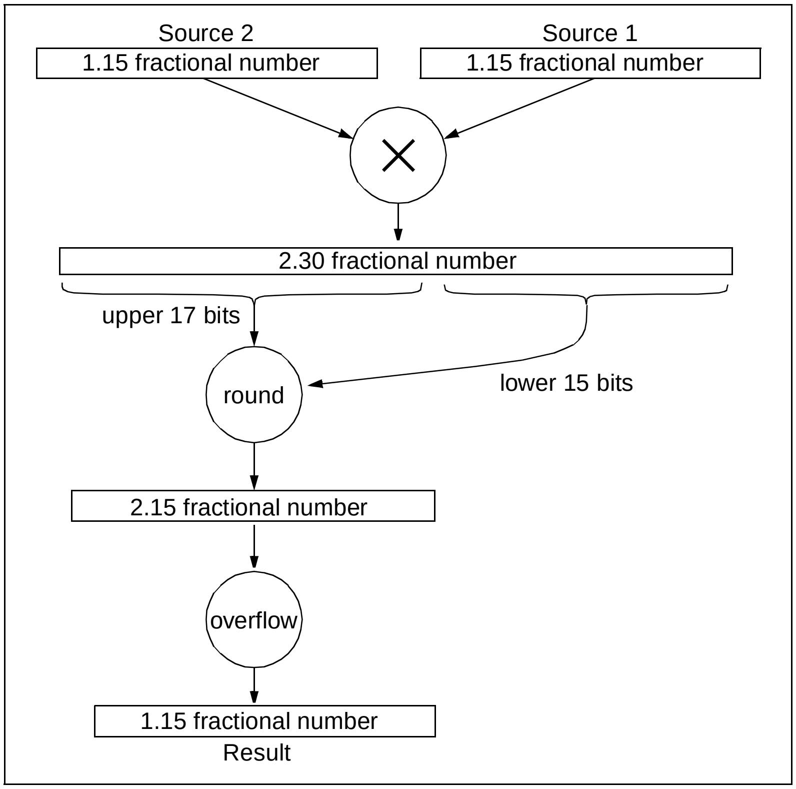


Figure 39: 16-Bit fractional rounding multiply

7.15 Multimedia multiply and sum

This instruction performs full-width multiplications on packed, signed, 16-bit elements, before summing the 32-bit intermediate results together and adding the result to an accumulation. The accumulator is a 64-bit scalar value, and the accumulation is performed using 64-bit modulo addition.

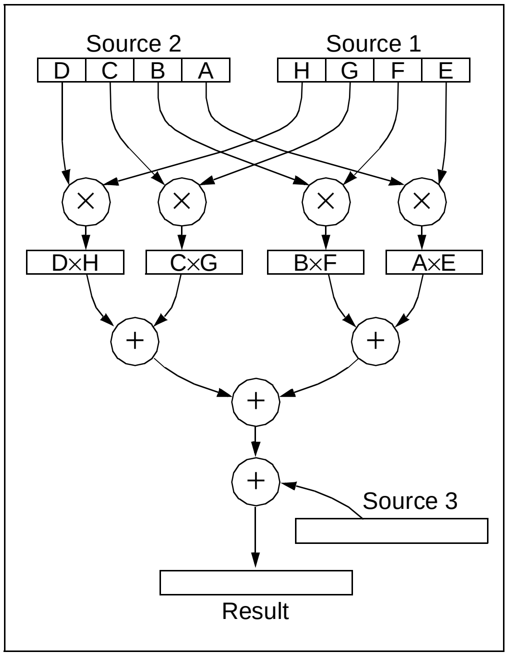


Figure 40: Multimedia 16-bit multiply sum

The same register operand specifies both source 3 and the result.

Since no precision can be lost in this operation, it can be applied to both integer and fractional types, although the latter will require some shifting at the end of the accumulation phase to produce a correctly-formed fractional quantity. Note that the MSHARDS.Q instruction, described in *Section 7.9*, can be used to reduce 64-bit accumulations back to 16-bit form.

Instruction	Summary
MMULSUM.WQ source1,source2,source3_result	Multimedia multiply and sum signed 16-bit

Table 68: Multimedia multiply and sum

7.16 Multimedia fractional multiply accumulate

This is a packed version of a traditional DSP multiply accumulate. It performs full-width multiplications on the lower halves of packed, fractional, 16-bit elements, then sums the packed 32-bit intermediate result with an accumulator also in packed 32-bit form.

The multiplication result is left shifted by 1 to align its fixed point with that of the accumulator. In the special case where a multiplication of MOSTNEG by MOSTNEG occurs, the result, which would otherwise be outside the representable range, is saturated to MOSTPOS.

The summing can give 33 bits of result for each accumulation, so a final saturation to 32 bits is performed on each half to give a properly formed result in packed, signed, fractional 32-bit format.

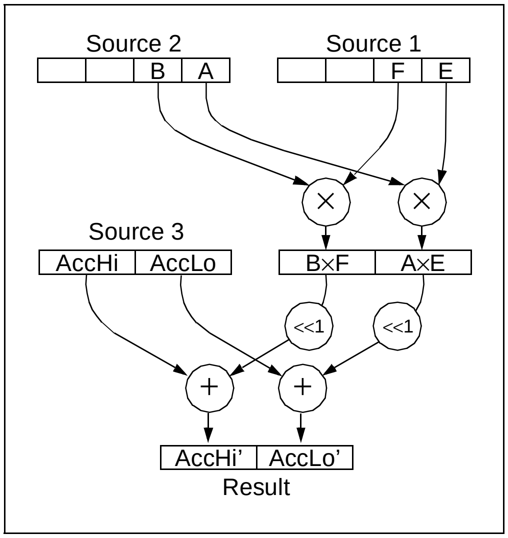


Figure 41: Multimedia 16-bit fractional multiply accumulate

The same register operand specifies both source 3 and the result.

Instruction	Summary
MMACFX.WL source1,source2,source3_result	Multimedia fractional multiply and accumulate signed 16-bit with saturation

Table 69: Multimedia fractional multiply accumulate

7.17 Multimedia fractional multiply subtract

This is a packed version of a traditional DSP multiply accumulate with negation. It performs full-width multiplications on the lower halves of packed, fractional, 16-bit elements, then subtracts the packed 32-bit intermediate result from an accumulator also in packed 32-bit form.

The multiplication result is left shifted by 1 to align its fixed point with that of the accumulator. In the special case where a multiplication of MOSTNEG by MOSTNEG occurs, the result, which would otherwise be outside the representable range, is saturated to MOSTPOS.

The subtraction can give 33 bits of result for each accumulation, so a final saturation to 32 bits is performed on each half to give a properly formed result in packed, signed, fractional 32-bit format.

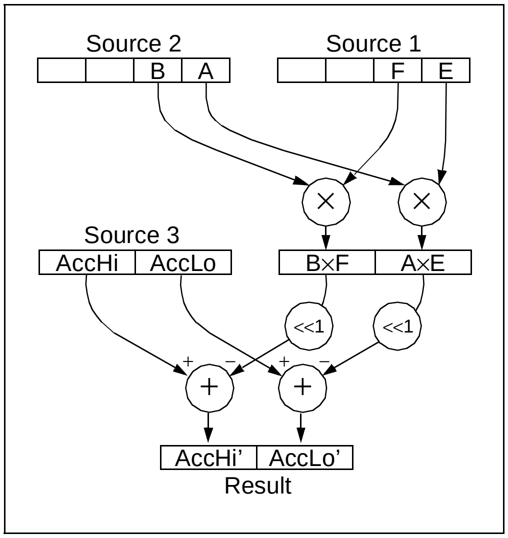


Figure 42: Multimedia 16-bit fractional multiply subtract

The same register operand specifies both source 3 and the result.

Instruction	Summary
MMACNFX.WL source1,source2,source3_result	Multimedia fractional multiply and subtract signed 16-bit with saturation

Table 70: Multimedia fractional multiply subtract

7.18 Multimedia shuffles

These instructions provide a set of perfect shuffles on all packed data widths. These can be used for conversions between different packed data formats, transposing matrices, rotating arrays and FFT butterflies.

The shuffles interleave two vectors, each of 64-bits in total, and hence the results are 128-bits long. Since the architecture has 64-bit registers, the shuffle instructions are defined in pairs, with one to produce the low half of the 128-bit result, and another to produce the high half.

There are 3 shuffles operations, one for each of the 3 packed element sizes: 8-bit, 16-bit and 32-bit. Each shuffle needs 2 instructions, one for the low half and one for the high half. The 3 shuffle operations and 6 required instructions are shown below.

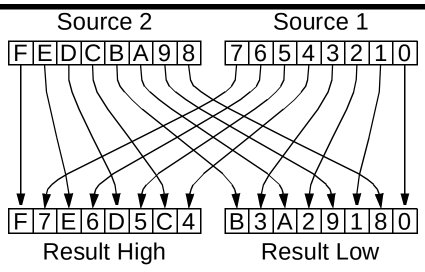
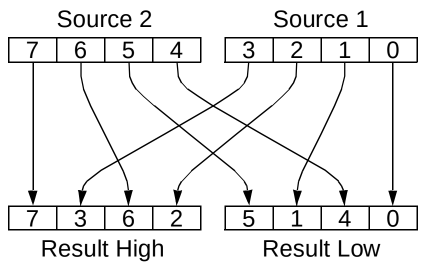
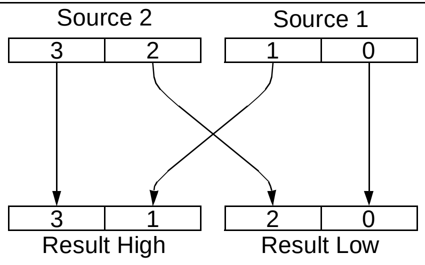
Data format	Shuffles	High half	Low half
8-bit (byte)	 Source 2 Source 1 F E D C B A 9 8 7 6 5 4 3 2 1 0 F 7 E 6 D 5 C 4 B 3 A 2 9 1 8 0 Result High Result Low	MSHFHI.B	MSHFLO.B
16-bit (word)	 Source 2 Source 1 7 6 5 4 3 2 1 0 7 3 6 2 5 1 4 0 Result High Result Low	MSHFHI.W	MSHFLO.W
32-bit (long-word)		MSHFHI.L	MSHFLO.L

Figure 43: Multimedia shuffles

Instruction	Summary
MSHFHI.B source1,source2,result	Multimedia shuffle upper-half 8-bit
MSHFLO.B source1,source2,result	Multimedia shuffle lower-half 8-bit
MSHFHI.W source1,source2,result	Multimedia shuffle upper-half 16-bit
MSHFLO.W source1,source2,result	Multimedia shuffle lower-half 16-bit
MSHFHI.L source1,source2,result	Multimedia shuffle upper-half 32-bit
MSHFLO.L source1,source2,result	Multimedia shuffle lower-half 32-bit

Table 71: Multimedia shuffle instructions

7.19 Multimedia bitwise conditional move

This instruction performs a bitwise conditional move from the source into the destination based on the value provided in a mask.

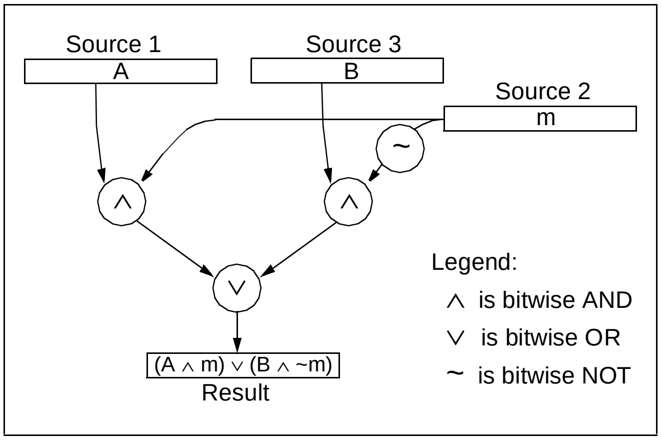


Figure 44: Bitwise conditional move

The same register operand specifies both source 3 and the result.

Instruction	Summary
MCMV source1,source2,source3_result	Multimedia bitwise conditional move

Table 72: Multimedia bitwise conditional move

7.20 Multimedia permute

This instruction performs 16-bit element permutations. For each 16-bit field in the result, two bits from the control operand determine which 16-bit field from the source is copied into that result field. *Figure 45* shows how field i (i=0,...,3) of the result is generated from the source operand. The instruction performs the operation for all four of the 16-bit fields in the result.

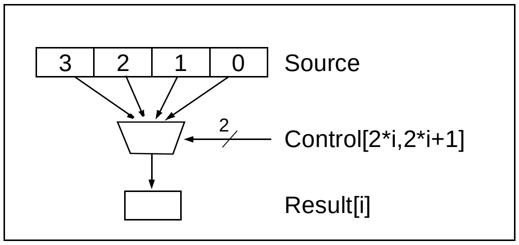


Figure 45: Permute

Instruction	Summary
MPERM.W source,control,result	Multimedia permute 16-bits

Table 73: Multimedia permute

This instruction can be used to replicate 16-bit and 32-bit fields throughout a packed vector. It can also be used to reverse the order of 16-bit or 32-bit fields. Many other permutations are possible. *Figure 46* shows some examples:

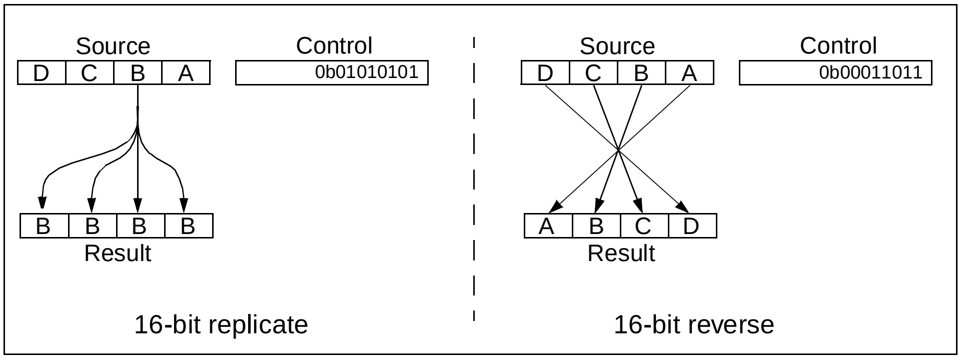


Figure 46: Permute examples

7.21 Multimedia extract

This instruction concatenates two 64-bit vector source operands together to form a 128-bit intermediate result, then extracts a contiguous 64-bit sub-vector from this at 8-bit offsets.

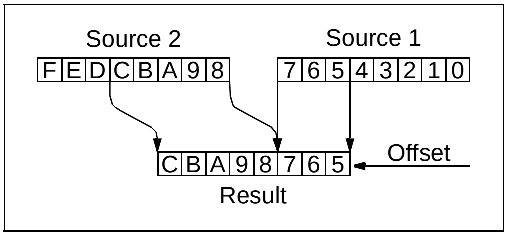


Figure 47: Extract operation

Extract is effectively a 4 operand instruction (two register sources, an immediate specifying the offset, and the result register). The architecture does not have instruction formats that directly support 4 operand instructions, and so extract is split into 7 different instructions, with the possible offsets of [1, 7] being represented in the opcode.

Instruction	Summary
MEXTR1 source1,source2,result	Multimedia extract 64 bits from 128 bits using a 1x8-bit offset
MEXTR2 source1,source2,result	Multimedia extract 64 bits from 128 bits using a 2x8-bit offset
MEXTR3 source1,source2,result	Multimedia extract 64 bits from 128 bits using a 3x8-bit offset
MEXTR4 source1,source2,result	Multimedia extract 64 bits from 128 bits using a 4x8-bit offset
MEXTR5 source1,source2,result	Multimedia extract 64 bits from 128 bits using a 5x8-bit offset
MEXTR6 source1,source2,result	Multimedia extract 64 bits from 128 bits using a 6x8-bit offset
MEXTR7 source1,source2,result	Multimedia extract 64 bits from 128 bits using a 7x8-bit offset

Table 74: Multimedia extract

SuperH

SHmedia floating-point 8

8.1 Introduction

SHmedia provides a comprehensive set of floating-point instructions for single-precision and double-precision representations. The SHmedia floating-point instructions are the set of instructions that are defined in this chapter. The SHmedia floating-point state consists of the SHmedia floating-point register set and FPSCR. These are defined in *Chapter 2: Architectural state on page 13*.

The IEEE754 floating-point standard is supported through a combination of hardware and system software. This is described in *Section 8.3: IEEE754 floating-point support on page 136*.

The architecture also provides non-IEEE754 support including fast handling of denormalized numbers, fused multiply accumulate and special-purpose instructions. This is described in *Section 8.4: Non-IEEE754 floating-point support on page 145*.

8.2 Floating-point disable

The architecture allows the floating-point unit to be disabled by software. This is achieved by setting SR.FD to 1. Once disabled, any attempt to execute a floating-point opcode will generate an exception. The set of floating-point opcodes is described in *Volume 2, Appendix A: SHmedia instruction encoding*.

If an implementation does not provide a floating-point unit, then the behavior of floating-point instructions is the same as an implementation with a floating-point unit that is permanently disabled. On such an implementation, SR.FD is permanently set to 1 and the implementation does not provide the floating-point state. Any attempt to access the floating-point state will generate an exception. It is possible to emulate the floating-point instructions and state in software.

8.3 IEEE754 floating-point support

The architecture supports IEEE754 floating-point. This is achieved through a combination of hardware and system software. The hardware provides a comprehensive set of floating-point instructions. These support standard floating-point data types and implement the most frequently required IEEE754 behavior. This allows a simple and high-performance hardware implementation.

This section defines which parts of the IEEE754 standard are provided in hardware. For systems requiring complete IEEE754 behavior, the remaining cases can be provided in system software. All details of such system software are strongly dependent on the software environment, and this is beyond the scope of this CPU architecture manual. Note that the term ‘system software’ is used here to mean software that is provided with the system, and does not necessarily imply that the software is part of an operating system.

The IEEE754 floating-point standard is defined in:

IEEE Standard for Binary Floating-point Arithmetic,
ANSI/IEEE Std. 754-1985.

It is assumed that the reader is familiar with the terminology used in this standard.

8.3.1 Formats

The architecture supports both IEEE754 basic formats: the 32-bit single format and the 64-bit double format. The encodings of these formats are described in *Section 3.4: IEEE754 floating-point numbers on page 33* and follow the IEEE754 standard.

Note that the architecture also supports the single extended format. This is implemented identically to the 64-bit double format. The architecture does not support the double extended format.

8.3.2 Rounding

The architecture provides hardware support for the round toward nearest and round toward zero rounding modes. There is no hardware support for round toward +∞ nor for round toward -∞. These cases can be emulated in system software for systems requiring complete IEEE754 compliance.

For conversions from floating-point formats to integer formats, hardware support is provided for only round toward zero. There is no hardware support for round toward nearest for these conversions. These can be provided in system software for systems requiring complete IEEE754 compliance.

The architecture provides the normal behavior of rounding results to the precision of the destination format. Thus, single-precision results are rounded to single-precision and held in single format, while double-precision results are rounded to double-precision and held in double format. This contrasts with some other systems that deliver results only to double or extended destinations.

8.3.3 Hardware operations

The architecture provides instructions for the following operations:

- Arithmetic: addition, subtraction, multiplication and division. Instructions are provided for both floating-point formats that perform these operations on 2 sources of that format yielding a result of the same format. Mixed format arithmetic can be synthesized by converting the narrower-format source to that of the wider-format source (this is an exact conversion), and using the appropriate wider-format operation.
- Square root extraction. Instructions are provided for both floating-point formats that perform this operation on a source of either format yielding a result of the same format.
- Conversion between different floating-point formats. Instructions are provided that convert, in either direction, between the two supported floating-point formats.
- Conversion between floating-point and integer formats. The supported integer formats for these conversions are signed 32-bit numbers and signed 64-bit numbers. Instructions are provided that convert, in either direction, between the two supported floating-point formats and these two signed integer formats.
- Compares: a sufficient set of compare instructions is provided. The full set can be synthesized using short instruction sequences. Instructions are provided for both floating-point formats that perform these operations on 2 sources of that format yielding a boolean result. Mixed format compares can be synthesized by converting the narrower-format source to that of the wider-format source (this is an exact conversion), and using the appropriate wider-format operation.

The architecture also provides instructions to copy floating-point values without change of format. These are not considered floating-point operations, and do not trigger any of the IEEE754 exceptional conditions.

8.3.4 Software operations

The following operations must be provided in system software for complete IEEE754 compliance:

- Arithmetic: remainder.
- Round to an integer value in floating-point format. This operation takes a floating-point value and rounds it to an integral-valued floating-point number in the same format.
- Convert between binary floating-point numbers and decimal strings.

For example, these operations could be provided in the form of compiler intrinsics, as a software library, or in some other form.

8.3.5 Zeroes, infinities, NaNs and sign

The IEEE754 standard specifies the bit-patterns for zeroes and infinities, and specifies the behavior of arithmetic with zero and infinite sources. These numbers are signed and the standard specifies the interpretation of the sign. The architecture supports zeroes and infinities exactly as required by the standard.

The IEEE754 standard distinguishes signaling NaNs and quiet NaNs, and specifies ranges of values that correspond to these categories of NaN. The standard does not place any interpretation on the sign of a NaN.

The architecture supports NaNs in a manner that complies with the standard. The architecture distinguishes source values as signaling NaNs or quiet NaNs as dictated by the standard. For destination values, the architecture always uses the representations for quiet NaNs given in *Table 75 Quiet NaN values*. This means that it is possible for an operation to have one (or more) quiet NaN inputs, and for the output to be a quiet NaN with a different representation.

Format	Quiet NaN value
Single	0x7FBFFFFF
Double	0x7FF7FFFF_FFFFFFFF

Table 75: Quiet NaN values

There are no cases where the architecture sets a destination to a signaling NaN value. The selection and interpretation of signaling NaN values is left to system software. The architecture chooses to not signal an IEEE754 exceptional condition when a signaling NaN is copied without a change of format.

The behavior for NaNs complies with IEEE754.

8.3.6 Exceptional conditions

The architecture supports an exception mechanism which, in combination with appropriate system software, is compliant with IEEE754. The terminology used here differs from the IEEE754 standard:

- An ‘exceptional condition’ occurs when an arithmetic operation detects an exceptionally unusual case. The equivalent IEEE754 term is ‘exception’.
- When an exceptional condition is detected, it is signaled to the user. This signaling is accomplished by the setting of status bits. It can also cause an ‘exception’ to be raised using the standard exception launch mechanism. The equivalent IEEE754 term is ‘trap’.
- Exception launch causes the execution of system software before being passed onto an ‘exception handler’. The equivalent IEEE754 term is ‘trap handler’.

The IEEE754 standard defines 5 exceptional conditions: invalid operation, division by zero, overflow, underflow and inexact. Each of these exceptional conditions is associated with the following state:

- A cause bit: this bit is set by the hardware if this instruction has detected this particular exceptional condition. This bit is automatically cleared by the hardware before the execution of any floating-point instruction which has the potential to raise any of the 5 IEEE754 exceptional conditions.
- A flag bit: this bit is set by the hardware if this instruction has detected this particular exceptional condition. This bit is only cleared at the request of the user. The flag bit acts as a ‘sticky’ bit recording any and all instances of this particular exceptional condition, across sequences of instructions delineated by the user.
- An enable bit: this bit allows the user to control whether detection of this particular exceptional condition raises an exception or not.

There are 3 bits for each of the 5 exceptional conditions giving a total of 15 bits used to support the IEEE754 exceptional conditions. The user can test and alter these bits individually, and can save and restore them together.

When an instruction raises an exception, no result is written. System software should arrange for the correct value to be written to the result, where required for IEEE754 compliance.

The distinction between the terms ‘signal’ and ‘raise’ is important. The detection of an exceptional condition is always signaled to the user through the setting of the relevant cause and enable bits. However, whether this signaling causes an exception to be raised depends upon the relevant enable bit.

Invalid operation

The architecture detects the standard IEEE754 invalid operations:

- Any operation on a signaling NaN. A copy without a change of format does not constitute an operation.
- Addition of differently signed infinities or subtraction of similarly signed infinities.
- Multiplication of a zero by an infinity.
- Division of a zero by a zero, or of an infinity by an infinity.
- Square root of numbers less than zero (note that this does not include the square root of -0, which is a valid operation yielding -0).
- Conversion of a floating-point number to an integer format where the floating-point number is an infinity, a NaN or overflows the integral range of the result format.
- Comparisons, which are neither unordered nor equality, where either source operand is a NaN.

An invalid operation is signaled when one of these conditions is detected.

If exceptions are enabled for invalid operations, then an exception is also raised. If no exception is raised and the destination has a floating-point format, the result is a quiet NaN. If no exception is raised and the destination has an integer format, then the result is either the most positive or most negative representable integer chosen according to the sign of the non-representable result. If no exception is raised and the destination has a boolean format, then the result is the appropriate boolean outcome of the comparison.

Remainder is not implemented in hardware. The remainder exceptional cases occur when the dividend is infinite or the divisor is zero. System software is responsible for signaling these cases.

Division by zero

The architecture detects the standard IEEE754 division by zero. A division by zero is signaled when the divisor is zero and the dividend is a finite non-zero number.

If exceptions are enabled for divide by zero, then an exception is also raised. If no exception is raised, the result is an appropriately signed infinity.

Note that division of an infinity by a zero is not an exceptional condition. It results in an appropriately signed infinity.

Overflow

An overflow condition is signaled when the rounded floating-point result, calculated as if the exponent range were unbounded, exceeds the largest representable finite number in the destination floating-point format.

If exceptions are enabled for overflow, then an exception is also raised. If no exception is raised, the result depends on the rounding mode:

- If the rounding mode is round toward nearest, then the result is an infinity with the same sign as the precise result.
- If the rounding mode is round toward zero, then the result is the largest representable finite number in the destination format with the same sign as the precise result.

When overflow exceptions are enabled, the architecture raises overflow exceptions on all instructions capable of overflow regardless of whether the overflow exception is signaled. System software is required to give the IEEE754 behavior. This is described in *Section 8.3.8*.

Underflow

An underflow condition arises due to tininess and loss of accuracy:

- Tininess results from the creation of a tiny non-zero result between $\pm 2^{Emin}$. These tiny numbers can lead to subsequent exceptional conditions; for example an overflow upon division.

  The architecture detects tininess after rounding. Tininess occurs when the result, calculated as if the exponent range were unbounded, is non-zero and lies strictly between $\pm 2^{Emin}$ where E_{min} is -126 for single format and -1022 for double format.
- Loss of accuracy is detected when the result cannot be represented exactly. It occurs when the result differs from which would have been calculated were both the exponent range and the precision unbounded. Loss of accuracy is the same as the inexact condition described in *Inexact on page 142*.

Underflow is signaled when both tininess and loss of accuracy are detected. Tininess or loss of accuracy alone are not sufficient to signal underflow.

If exceptions are enabled for underflow, then an exception is also raised. If no exception is raised, the result can be a zero, a denormalized number or $\pm 2^{Emin}$ as required by IEEE754 arithmetic.

When underflow exceptions are enabled, the architecture raises underflow exceptions on all instructions capable of underflow regardless of whether the underflow exception is signaled. System software is required to give the IEEE754 behavior. This is described in *Section 8.3.8*.

The IEEE754 standard states that when underflow exceptions are enabled and tininess is detected, then underflow is signaled regardless of whether there is loss of accuracy. This behavior is not provided directly in hardware, and system software is required to give the IEEE754 behavior.

Inexact

An inexact condition is signaled when the rounded result of an operation differs from the exact result or when the result has overflowed.

If exceptions are enabled for inexact conditions, then an exception is also raised. If no exception is raised, the result is the expected rounded or overflowed value.

When inexact exceptions are enabled, the architecture raises inexact exceptions on all instructions capable of inexact regardless of whether the inexact exception is signaled. System software is required to give the IEEE754 behavior. This is described in *Section 8.3.8*.

For IEEE754 behavior, inexact should not be signaled when an overflow exception is raised. This behavior is not provided directly in hardware, and system software is required to give the IEEE754 behavior.

8.3.7 Denormalized numbers

The architecture supports IEEE754 denormalized numbers through a combination of hardware and system software.

When the hardware performs a floating-point arithmetic operation that yields a denormalized result, the hardware produces the correct denormalized result as required by the IEEE754 standard.

When a denormalized number is encountered as a source to the computational floating-point operations defined below, an exception (called an FPU Error) is raised. FPU Error is associated with a cause bit to indicate when it occurs. Neither a flag bit nor an enable bit is provided.

The FPU Error exception is raised only for denormalized sources. It can be readily distinguished from other exception cases, and is not raised in combination with other exception cases. System software is required to emulate the required IEEE754 behavior for computational operations involving source denormalized numbers. This emulation should include any further IEEE754 exceptions resulting from that operation.

The IEEE754-compliant computational operations which can raise FPU Error are:

- Arithmetic: addition, subtraction, multiplication and division.
- Square root extraction.
- Conversion between different floating-point formats.

If a dyadic floating-point instruction has a denormalized source and a NaN source (either signaling or quiet), then there is no FPU Error. Instead, the instruction signals an invalid operation (for a signaling NaN source), or produces a quiet NaN result. In such cases the denormalized source has no effect on the behavior.

The hardware fully supports denormalized sources for floating-point to integer conversions, compares and copies without change of format.

If IEEE754 compliant handling of denormalized numbers is not required, then a mode is provided which forces both source and result denormalized numbers to an appropriately signed zero. This is known as flush-to-zero and is described in *Section 8.4.1*.

8.3.8 Exception launch

The architecture provides hardware that can raise an exception for all cases required by the IEEE754 standard. Appropriate system software can use these exceptions to call an exception handler with the facilities indicated by IEEE754.

The only combinations of exception that can be signaled together are overflow with inexact and underflow with inexact. In fact, the floating-point architecture is arranged such that overflow and underflow are always accompanied by inexact. Note that it is possible for inexact to be signaled without either overflow or underflow occurring.

If multiple floating-point exceptions are raised by the execution of a particular floating-point instruction, then one exception handler is launched. The cause and flag bits are updated for all detected exceptions.

The overflow, underflow and inexact exceptions are raised ‘early’ by the hardware before it is known whether those exceptional conditions will arise. This means that if one or more of these exceptions are enabled, then a floating-point instruction which has the potential to raise one of those enabled exceptions will always raise an exception, regardless of whether that condition actually arises with the provided source operands. The cause and flag bits are still updated accurately and their correct values will be delivered through the launch mechanism. System software can take appropriate action to provide the required IEEE754 behavior.

8.3.9 Recommended functions and predicates

The architecture provides additional hardware support consistent with the IEEE754 recommendations:

- Instructions are provided to negate a source operand in either of the supported floating-point formats. This simply reverses the sign bit. This is not considered an arithmetic operation, and does not signal invalid operations.
- Instructions are provided for unordered comparison of two source operands both of which are in one of the supported floating-point formats.

The remaining IEEE754 recommended functions and predicates should be implemented in software if required.

Additionally, the architecture provides instructions to take the absolute value of a source operand in either of the supported floating-point formats. This simply clears the sign bit. This is not considered an arithmetic operation, and does not signal invalid operations.

8.3.10 Future FPU architecture

The architecture retains the option of extending the IEEE754 floating-point support in future versions. Specifically:

- The supported rounding modes could be extended to include rounding toward $+\infty$ and rounding toward $-\infty$.
- The overflow, underflow and inexact exceptional conditions could be detected ‘late’ so that exceptions are only raised when these exceptional conditions are signaled. The effect of this potential architectural extension is that the overflow, underflow and inexact exceptions are raised only when required by the IEEE754 standard, and that other cases that currently raise these exceptions give the correct IEEE754 result without the intervention of system software.
- The IEEE754 behavior for denormalized source operands could be provided in hardware. The effect of this potential architectural extension is that the FPU Error

exception is not raised, and that all instructions with denormalized source operands give the correct IEEE754 result without the intervention of system software.

Software must not exploit the current FPU architecture in a way that would be incompatible with these potential future extensions. In particular, software must not exploit the current definitions of the overflow, underflow, inexact and FPU Error exceptions in a way that would be incompatible with the future extensions described above.

The information in this section does not require future architecture versions to use these options, nor does it constrain future architecture versions to just these options.

8.4 Non-IEEE754 floating-point support

The architecture supports additional floating-point support beyond that required for IEEE754 conformance. The following features should not be used where strict IEEE754 behavior is required.

8.4.1 Treat denormalized numbers as zero

When FPSCR.DN is set to 1, denormalized numbers are not handled according to the IEEE754 standard. Instead, negative denormalized numbers are treated as -0, and positive denormalized numbers are treated as +0.

For denormalized source values, this conversion occurs before exceptional conditions and special cases are handled. Thus the instruction executes exactly as if any denormalized source values were an appropriately signed zero.

For denormalized result values, the conversion occurs as a final stage of processing after the result has been calculated according to IEEE754. Additionally, when a denormalized result is flushed-to-zero, inexact and underflow conditions are signaled to denote the loss of precision.

8.4.2 Fused multiply accumulate support

A single-precision floating-point multiply-accumulate instruction (FMAC.S) is provided. The multiplication and addition are performed as if the exponent and precision ranges were unbounded, followed by one rounding down to single-precision format. This algorithm is called a fused-mac and is typically implemented in fewer cycles than a separate multiply and addition.

The results of this instruction can differ from those which would be generated by a separate multiply followed by an addition:

- The fused-mac algorithm produces more precise results than using a multiply followed by an addition. This because the fused-mac algorithm uses a single-rounding down to single precision, as opposed to the two roundings used in a multiply and addition sequence.
- Special case detection can give different behavior. The multiply-accumulate detects special cases through analysis of the three input operands. A multiply and addition sequence detects special cases on the multiplication sources, and then on the addition sources. Consider a case where the multiplication sources are finite, but where the multiplication result would overflow to positive infinity if converted to a single-precision format. The multiply-accumulate will compute a fully precise intermediate, rather than an infinity, and will not cause an invalid operation should this intermediate be added to an oppositely signed infinity.

8.4.3 Special-purpose instructions

Special-purpose instructions are provided to accelerate certain applications where strict IEEE754 conformance is not required. These instructions are:

- FIPR.S: approximate single-precision floating-point inner product.
- FTRV.S: approximate single-precision floating-point matrix transformation.
- FCOSA.S: approximate single-precision floating-point cosine.
- FSINA.S: approximate single-precision floating-point sine.
- FSRRA.S: approximate single-precision floating-point reciprocal of a square root.

These instructions yield results that can differ from the exact results within an architecturally specified bound. Results on different implementations can vary within these allowed bounds, though any particular implementation gives deterministic results for given sources. These instructions signal the inexact condition to denote potential loss of precision.

Further details of these instructions are given in *Section 8.7: Special-purpose floating-point instructions on page 154* and *Volume 2 Chapter 2: SHmedia instruction set*.

8.5 Floating-point status and control register

The floating-point status and control register, FPSCR, is 32 bits wide. It is accessed using the FGETSCR and FPUTSCR instructions. The SHmedia view of FPSCR is shown in *Figure 48*. The ‘r’ field indicates reserved bits.

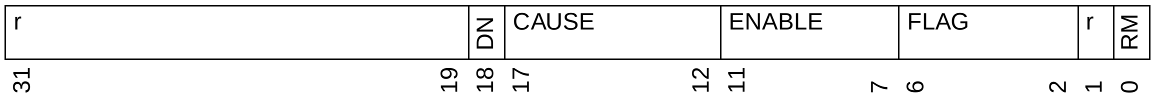


Figure 48: FPSCR register

The interpretation of each field is given in the following table.

Field name	FPSCR bit	Behavior
RM	0	If 0x0: round to nearest. If 0x1: round to zero
FLAG	2 3 4 5 6	Sticky flag for inexact exceptions (FLAG.I) Sticky flag for underflow exceptions (FLAG.U) Sticky flag for overflow exceptions (FLAG.O) Sticky flag for divide by zero exceptions (FLAG.Z) Sticky flag for invalid exceptions (FLAG.V)
ENABLE	7 8 9 10 11	Enable flag for inexact exceptions (ENABLE.I) Enable flag for underflow exceptions (ENABLE.U) Enable flag for overflow exceptions (ENABLE.O) Enable flag for divide by zero exceptions (ENABLE.Z) Enable flag for invalid exceptions (ENABLE.V)
CAUSE	12 13 14 15 16 17	Cause flag for inexact exceptions (CAUSE.I) Cause flag for underflow exceptions (CAUSE.U) Cause flag for overflow exceptions (CAUSE.O) Cause flag for divide by zero exceptions (CAUSE.Z) Cause flag for invalid exceptions (CAUSE.V) Cause flag for FPU error exceptions (CAUSE.E)
DN	18	If 0: a denormalized source operand raises an FPU error exception If 1: a denormalized source operand is flushed to zero before the floating-point operation is performed, and a denormalized result is flushed to zero after the floating-point operation is performed

Table 76: FPSCR fields

Field name	FPSCR bit	Behavior
Reserved (r)	1 and 19 to 31	Write reserved bits as zero; ignore value read from reserved bits

Table 76: FPSCR fields

8.6 General-purpose floating-point instructions

This section describes the general-purpose floating-point instruction set. This includes:

- Floating-point status and control
- Floating-point dyadic arithmetic
- Floating-point monadic arithmetic
- Floating-point comparisons
- Floating-point conversions
- Floating-point multiply-accumulate
- Floating-point moves

8.6.1 Floating-point status and control

Two instructions are used to access the status of the floating-point unit. The instructions operate on a user-visible status and control register called FPSCR, described in *Section 8.5*. Instructions are provided to transfer a value from a single-precision floating-point register into FPSCR, and to transfer a value from FPSCR into a single-precision floating-point register.

Instruction	Summary
FPUTSCR source	move to floating-point status/control register
FGETSCR result	move from floating-point status/control register

Table 77: Floating-point status and control instructions

Separate instructions are provided to transfer values between floating-point registers and general-purpose registers (see *Section 8.6.7: Floating-point moves on page 153*).

8.6.2 Floating-point dyadic arithmetic

This set of floating-point instructions operate on two floating-point source operands. The following instructions are defined:

Instruction	Summary
FADD.S source1,source2,result	add two single-precision numbers
FADD.D source1,source2,result	add two double-precision numbers
FSUB.S source1,source2,result	subtract two single-precision numbers
FSUB.D source1,source2,result	subtract two double-precision numbers
FMUL.S source1,source2,result	multiply two single-precision numbers
FMUL.D source1,source2,result	multiply two double-precision numbers
FDIV.S source1,source2,result	divide two single-precision numbers
FDIV.D source1,source2,result	divide two double-precision numbers

Table 78: Floating-point dyadic arithmetic instructions

8.6.3 Floating-point monadic arithmetic

This set of floating-point instructions operate on one floating-point source operand. The following instructions are defined:

Instruction	Summary
FABS.S source,result	get absolute value of a single-precision number
FABS.D source,result	get absolute value of a double-precision number
FNEG.S source,result	negate a single-precision number
FNEG.D source,result	negate a double-precision number
FSQRT.S source,result	find square root of a single-precision number
FSQRT.D source,result	find square root of a double-precision number

Table 79: Floating-point monadic arithmetic instructions

The absolute and negate operations are not considered as arithmetic operations and they do not generate a floating-point exception.

8.6.4 Floating-point multiply-accumulate

A floating-point multiply and accumulate instruction is defined to multiply two source operands and then add this to a third source operand. The third source operand also defines the destination into which the result is placed. The effect is that the output of the multiplication is accumulated into the destination register.

The multiplication and addition are performed as if the exponent and precision ranges were unbounded, followed by one rounding down to single-precision format. This algorithm is called a fused-mac. Where IEEE754 compliance is required, the programmer should use separate multiply and add instructions.

Instruction	Summary
FMAC.S source1,source2,source3_result	single-precision fused multiply accumulate

Table 80: Floating-point multiply-accumulate

8.6.5 Floating-point conversions

A set of conversion instructions are defined to convert between the two floating-point formats and between integer and floating-point formats:

Instruction	Summary
FCNV.SD source,result	single-precision to double-precision conversion
FCNV.DS source,result	double-precision to single-precision conversion
FTRC.SL source,result	single-precision to 32-bit integer conversion
FTRC.DL source,result	double-precision to 32-bit integer conversion
FTRC.SQ source,result	single-precision to 64-bit integer conversion
FTRC.DQ source,result	double-precision to 64-bit integer conversion
FLOAT.LS source,result	32-bit integer to single-precision conversion
FLOAT.LD source,result	32-bit integer to double-precision conversion
FLOAT.QS source,result	64-bit integer to single-precision conversion
FLOAT.QD source,result	64-bit integer to double-precision conversion

Table 81: Floating-point conversion instructions

All instructions read inputs from floating-point registers and write results to floating-point registers. The integer to floating-point conversion instructions interpret their source as an integer value. The floating-point to integer conversions write an integer result.

All floating-point to integer conversions convert by truncation (also known as round to zero). The floating-point to 32-bit integer conversion saturates the output to maximum/minimum values in a signed 32-bit range. The floating-point to 64-bit integer conversion saturates the output to maximum/minimum values in a signed 64-bit range.

8.6.6 Floating-point comparisons

The following comparison instructions are defined:

Instruction	Summary
FCMPEQ.S source1,source2,result	compare single-precision numbers for equality
FCMPEQ.D source1,source2,result	compare double-precision numbers for equality
FCMPGT.S source1,source2,result	compare single-precision numbers for greater-than
FCMPGT.D source1,source2,result	compare double-precision numbers for greater-than
FCMPGE.S source1,source2,result	compare single-precision numbers for greater-or-equal
FCMPGE.D source1,source2,result	compare double-precision numbers for greater-or-equal
FCMPUN.S source1,source2,result	compare single-precision numbers for unorderedness
FCMPUN.D source1,source2,result	compare double-precision numbers for unorderedness

Table 82: Floating-point compare instructions

Two floating-point numbers, x and y, are related by exactly one of the following four mutually exclusive relations:

- x is less than y
- x is equal to y
- x is greater than y
- x and y are unordered

Unordered occurs when at least one of x or y is a NaN, even if they are the same NaN.

Table 83 shows the mapping from IEEE754 comparisons onto supported floating-point instructions. In this table, the result of the compare is stored as a boolean in a general-purpose register R_d.

Format	IEEE754 comparison (ad hoc notation)	Comparison description	Instruction
Single	FR_x = FR_y	Equal	FCMPEQ.S FR_x, FR_y, R_d
	FR_x > FR_y	Greater than	FCMPGT.S FR_x, FR_y, R_d
	FR_x >= FR_y	Greater than or equal	FCMPGE.S FR_x, FR_y, R_d
	FR_x ? FR_y	Unordered	FCMPUN.S FR_x, FR_y, R_d
Double	DR_x = DR_y	Equals	FCMPEQ.D DR_x, DR_y, R_d
	DR_x > DR_y	Greater than	FCMPGT.D DR_x, DR_y, R_d
	DR_x >= DR_y	Greater than or equal	FCMPGE.D DR_x, DR_y, R_d
	DR_x ? DR_y	Unordered	FCMPUN.D DR_x, DR_y, R_d

Table 83: Floating-point compares

The remaining IEEE754 comparisons can be synthesized using the equivalent sequences shown in *Table 84*. The same equivalences are applicable to both single-precision and double-precision operations.

IEEE754 comparison (ad hoc notation)	Comparison description	Equivalent sequence
x ?<> y	Not equal	NOT (x = y)
x < y	Less than	y > x
x <= y	Less than or equal	y >= x
x <> y	Less than or greater than	(x > y) OR (y > x)
x <=> y	Less than or equal or greater than	(x > y) OR (y >= x)
x ?> y	Unordered or greater than	(x ? y) || (x > y)
x ?>= y	Unordered or greater than or equal	(x ? y) || (x >= y)

Table 84: Equivalent floating-point compares

IEEE754 comparison (ad hoc notation)	Comparison description	Equivalent sequence
x ?< y	Unordered or less than	(x ? y) || (y > x)
x ?<= y	Unordered or less than or equal	(x ? y) || (y >= x)
x ?= y	Unordered or equal	(x ? y) OR (x = y)

Table 84: Equivalent floating-point compares

In *Table 84*, OR represents conventional boolean-or and NOT represents conventional boolean-not. The '||' notation is used to denote a short-circuited boolean-or, where the second operand is not evaluated if the first operand is true. This is necessary to avoid taking inappropriate invalid operation exceptions, and can be realized by branching around the evaluation of the second operand.

The 'ad hoc' notation used in *Table 83* and *Table 84*, and the set of comparisons described, correspond to those defined in the IEEE754 standard. The result of an IEEE754 comparison can be negated using an integer instruction to negate the boolean result. In some cases this negation can be folded into a branch or conditional move to give a better sequence.

8.6.7 Floating-point moves

A set of register-to-register moves are defined to transfer values from and to floating-point registers. These instructions transfer bit-patterns and do not interpret or modify the values transferred. They are not considered arithmetic instructions and do not generate any floating-point exception. Some of the following instructions transfer values between floating-point registers; some of them transfer values between a floating-point register and a general-purpose register.

Instruction	Summary
FMOV.S source,result	32-bit floating-point to floating-point register move
FMOV.D source,result	64-bit floating-point to floating-point register move
FMOV.SL source,result	32-bit floating-point to general register move
FMOV.DQ source,result	64-bit floating-point to general register move
FMOV.LS source,result	32-bit general to floating-point register move
FMOV.QD source,result	64-bit general to floating-point register move

Table 85: Floating-point move instructions

8.7 Special-purpose floating-point instructions

This section describes a set of instructions that are mainly used in special-purpose applications (for example, graphics). These instructions do not conform to the IEEE754 standard. Further details of these instructions are given in their specification in *Volume 2 Chapter 2: SHmedia instruction set*.

Instruction	Summary
FIPR.S source1,source2,result	compute inner (dot) product of two vectors
FTRV.S source1,source2,result	transform vector
FCOSA.S source,result	compute cosine of an angle
FSINA.S source,result	compute sine of an angle
FSRRA.S source,result	compute reciprocal of a square root of a value

Table 86: Special-purpose floating-point instructions

8.7.1 Mathematical properties

These instructions return approximate results. An implementation will return a result which is strictly within a specified error bound relative to the fully precise result. Thus, the absolute error in the result will be less than the error bound specified for that instruction. The approximate result computations allow faster and more cost-effective implementations of these instructions.

The actual error in the returned value of an approximate instruction is denoted as:

$$\mathtt{actual\_error} = |f_{\mathtt{ideal}} - f_{\mathtt{implementation}}|$$

Where f_{ideal} is the infinitely precise result and $f_{implementation}$ is the finite-precision floating-point value returned by a particular implementation of that instruction. The specified error, *spec_error*, defined for each approximate instruction, specifies a strict upper bound on the *actual_error*. An implementation of the architecture satisfies the following condition for all cases:

$$\mathtt{actual\_error} < \mathtt{spec\_error}$$

For a given set of operand values, a particular implementation of these instructions is guaranteed to give a deterministic result. This means that every time that an approximate instruction is executed on a particular implementation with a given set of operand values, then the same result will be calculated. However, a different implementation is not guaranteed to give the same result for that same set of operand values, though all implementations will produce results strictly within the architected error bound.

8.7.2 FIPR.S and FTRV.S calculation

FIPR.S and FTRV.S use vector and matrix data structures:

- A vector, FV_x, is defined as a collection of four single-precision floating-point registers in the following order:

  $FV_{4n} = \{FR_{4n}, FR_{4n+1}, FR_{4n+2}, FR_{4n+3}\}$, n = 0, 1, through to 15.

- A matrix, $MTRX_x$, is defined as a collection of sixteen single-precision floating-point registers:

  $MTRX_{16n} = \{FR_{16n}, FR_{16n+1},$ through to $FR_{16n+15}\}$, n = 0, 1, 2, 3.

The use of these data structures by FIPR.S and FTRV.S is shown in *Figure 49* and *Figure 50*. All operands of these instructions are single-precision floating-point.

FIPR.S takes two vectors FV_g and FV_h and performs a 4-element floating-point inner product to give one floating-point result which is placed in FR_f.

$$\begin{bmatrix} FR_{g+0} & FR_{g+1} & FR_{g+2} & FR_{g+3} \end{bmatrix} \times \begin{bmatrix} FR_{h+0} \\ FR_{h+1} \\ FR_{h+2} \\ FR_{h+3} \end{bmatrix} \text{->} \begin{bmatrix} FR_f \end{bmatrix}$$

Figure 49: FIPR.S

FTRV.S takes a matrix $MTRX_g$ and a vector FV_h and performs a matrix by vector multiplication to give a vector floating-point result which is placed in FV_f. The matrix uses a column-major ordering; this means that the element numbers in the matrix increment by 1 down a column, but by 4 across a row.

$$\begin{bmatrix} FR_{g+0} & FR_{g+4} & FR_{g+8} & FR_{g+12} \\ FR_{g+1} & FR_{g+5} & FR_{g+9} & FR_{g+13} \\ FR_{g+2} & FR_{g+6} & FR_{g+10} & FR_{g+14} \\ FR_{g+3} & FR_{g+7} & FR_{g+11} & FR_{g+15} \end{bmatrix} \times \begin{bmatrix} FR_{h+0} \\ FR_{h+1} \\ FR_{h+2} \\ FR_{h+3} \end{bmatrix} \text{->} \begin{bmatrix} FR_{f+0} \\ FR_{f+1} \\ FR_{f+2} \\ FR_{f+3} \end{bmatrix}$$

Figure 50: FTRV.S

FIPR.S and FTRV.S support denormalized numbers. When FPSCR.DN is 0, denormalized numbers are treated as their denormalized value in the calculation. When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. These instructions never signal an FPU error.

FTRV.S does not check all of its inputs for invalid operations and then raise an exception accordingly. If invalid operation exceptions are requested by the user, the FTRV.S instruction always raises that exception. If this exception is not requested by the user, then each of the four inner-products is checked separately for an invalid operation and the appropriate result is set to a quiet NaN for each inner-product that is invalid.

8.7.3 FIPR.S and FTRV.S accuracy specification

The FIPR.S instruction computes the dot-product of two vectors, FV_g and FV_h, and places the result in FR_f. Each vector contains four single-precision floating-point values: FV_g contains FR_g, FR_{g+1}, FR_{g+2} and FR_{g+3}, while FV_h contains FR_h, FR_{h+1}, FR_{h+2} and FR_{h+3}. The dot-product is specified as:

$$FR_f = \sum_{i=0}^{3} FR_{g+i} \times FR_{h+i}$$

The limited accuracy reduces the number of significant bits in the fraction of the result. The absolute amount of allowed error in the result depends on the exponent of the result. Additionally, the result is computed as a limited-precision sum of four intermediate values. This means that the absolute amount of allowed error also depends on the largest exponent of these four intermediates.

The accuracy specification is defined as follows. Firstly, define a notation to represent the biased exponent values of the input operands FR_{g+i} and FR_{h+i}:

eFR_{g+i} = biased exponent value of FR_{g+i}

eFR_{h+i} = biased exponent value of FR_{h+i}

These values are simply the values of ‘e’ for each floating-point source as defined in *Section 3.4: IEEE754 floating-point numbers on page 33*. These values will be in the range [1, 254].

The unbiased pre-normalized exponent of the product of FR_{g+i} and FR_{h+i} is now calculated. The pre-normalized exponent of this product is given simply by adding the two multiplicand exponents together. Since this is a pre-normalized exponent there is no need to consider the contribution of the multiplied fractions. The pre-normalized exponent is converted to an unbiased form by subtracting 127 twice, removing the biases of each multiplicand. This gives values in the range [-252, 254].

Additionally, if either multiplicand is zero the result is special-cased to a value of *EZ*. The value of *EZ* is selected to be easily distinguishable from other possible values: a value of -253 is convenient. The detection of multiplications by zero allows the error bound to be forced to zero in the special case where all addition terms are multiplications by zero. The calculation of the unbiased pre-normalized exponent EP_i of the product of FR_{g+i} and FR_{h+i} can now be specified as:

$$EP_i = \begin{cases} \mathrm{EZ} & \mathrm{if}((\mathrm{FR}_{g+i} = 0.0)\mathrm{OR}(\mathrm{FR}_{h+i} = 0.0)) \\ \max(\mathrm{eFR}_{g+i},1) + \max(\mathrm{eFR}_{h+i},1) - 254 & \mathrm{otherwise} \end{cases}$$

The specified error depends on the largest intermediate term to the 4-element addition. The largest value of EP_i where i is in [0, 3] is calculated as:

$$EPm = \max\,(EP_0, EP_1, EP_2, EP_3)$$

where ‘max’ computes the maximum of its operands. The value of *EZ* is less than all other possible values of EP_i. This means that *EPm* will only take the value of *EZ* if all 4 EP_i are *EZ*. The range of *EPm* is [-252, 254] if the special value *EZ* is excluded.

The specified error in the result value can now be defined as:

$$spec\_error = \begin{cases} 0 & \mathrm{if}(\mathrm{EPm} = \mathrm{EZ}) \\ 2^{\mathrm{EPm}-24} + 2^{\mathrm{E}-24+\mathrm{rm}} & \mathrm{if}(\mathrm{EPm} \neq \mathrm{EZ}) \end{cases}$$

The rounding mode is indicated by *rm*:

$$rm = \begin{cases} 0 & \mathrm{if(round-to-nearest)} \\ 1 & \mathrm{if(round-to-zero)} \end{cases}$$

E is the unbiased exponent of the actual result produced by the FIPR.S instruction. If the result is a non-zero normalized number, the value of *E* is the biased exponent of the result minus the exponent bias; this is (*e* - 127). If the result is a denormalized number, the value of E is E_{min}, which has the value -126. If the result is exactly a positive or negative zero, then E is also E_{min}. Thus, the range of E is [-126, 127].

In the special case where EPm is EZ the specified error is zero. Otherwise, the specified error is defined as the sum of two terms. The first term is calculated from the maximum, unbiased, pre-normalized exponent among the four addends. This term gives an error contribution of 2^{EPm-24}, resulting from 23 bits of precision in the calculation of the largest multiplication. The second term is calculated from the unbiased exponent of the actual FIPR.S result. This term gives an error contribution of $2^{E-24+rm}$, resulting from (23 - rm) bits of precision in the addition of the four addends and normalization back to the IEEE754 format. Round-to-nearest gives 23 bits of precision in this term, while this is reduced to 22 bits of precision for round-to-zero due to the increased amount of error introduced by this rounding. The total error is the sum of these two error contributions.

The FTRV.S instruction is numerically equivalent to performing 4 separate FIPR.S operations on the rows of a 4x4 matrix against the column of a 4x1 vector to give a 4x1 vector result. The specified error in each element of the result is computed using the same algorithm as that described for FIPR.S.

These instructions are not guaranteed to obey the standard mathematical notions of commutativity and associativity. This means that swapping the order of the source operands to a multiply or to an addition can cause the result to change. Furthermore, the association of the additions is not defined and they can be evaluated in any order chosen by the implementation.

8.7.4 FCOSA.S, FSINA.S and FSRRA.S

The FCOSA.S and FSINA.S instructions compute the cosine and sine (respectively) of an angle held in FR_g and place the result in FR_f. The input angle is the amount of rotation expressed as a signed fixed-point number in a 2's complement representation. The value 1 represents an angle of $360^o/2^{16}$. The upper 16 bits indicate the number of full rotations and the lower 16 bits indicate the remainder angle between 0^o and 360^o. The result is the cosine or sine of the angle in single-precision floating-point format.

The FSRRA.S instruction computes the reciprocal of the square root of the value held in FR_g and places the result in FR_f. The source and result of this instruction are both held in single-precision floating-point format.

These are approximate instructions. The specified error in the result value is:

$$\text{spec\_error} = 2^{E-21}$$

where E is the unbiased exponent value of result.

8.8 Floating-point memory instructions

Load and store instructions transfer data between floating-point registers and memory. Three different widths of data are supported:

- A suffix of '.S' indicates an instruction that transfers 32-bit data to a 4-byte aligned effective address. The floating-point operand FR_i, where i is in [0, 63], denotes one of 64 single-precision floating-point registers.
- A suffix of '.D' indicates an instruction that transfers 64-bit data to an 8-byte aligned effective address. The floating-point operand DR_{2i}, where i is in [0, 31], denotes one of 32 double-precision floating-point registers.
- A suffix of '.P' indicates an instruction that transfers a pair of two 32-bit data values to an 8-byte aligned effective address. The floating-point operand FP_{2i}, where i is in [0, 31], denotes one of 32 pairs of single-precision floating-point registers.

The behavior of '.D' and '.P' transfers is different due to endianness. A full description of the behavior is given in *Section 3.7: Data representation in memory on page 43*. Floating-point load and store instructions are provided for naturally aligned data only. If access to misaligned data is required, the integer load and store instructions for misaligned access should be used, with an appropriate move instruction to transfer the data between the floating-point and general-purpose register sets.

The floating-point load and store instructions are typically used for transferring floating-point data. However, these instructions place no interpretation on the values that they transfer. They can also be used to load and store integer values into the floating-point registers. This can be convenient since some instructions (notably conversions to and from integers) interpret floating-point registers as integers.

The two supported addressing modes are displacement and indexed. The supported instructions are summarized in the following table.

Access	Mode	Single-precision	Double-precision	Single-precision pair
Load	indexed	FLDX.S	FLDX.D	FLDX.P
	displacement	FLD.S	FLD.D	FLD.P
Store	indexed	FSTX.S	FSTX.D	FSTX.P
	displacement	FST.S	FST.D	FST.P

Table 87: Aligned floating-point load and store instructions

8.8.1 Displacement addressing

For displacement addressing, the effective address is calculated by adding a base pointer to a displacement. The base register is held in a general-purpose register, and the displacement is given as a sign-extended 10-bit immediate value. The immediate value is scaled by the size of the object accessed.

Instruction	Summary	Displacement scaling factor
FLD.S base,offset,result	load 32-bit value	4
FLD.D base,offset,result	load 64-bit value	8
FLD.P base,offset,result	load two 32-bit values	8
FST.S base,offset,value	store 32-bit value	4
FST.D base,offset,value	store 64-bit value	8
FST.P base,offset,value	store two 32-bit values	8

Table 88: Aligned floating-point load and store instructions with displacement addressing

8.8.2 Indexed addressing

For indexed addressing, the effective address is calculated by adding a base pointer with an index. Both the base pointer and the index are held in general-purpose registers. Unlike displacement addressing, the index is not scaled.

Instruction	Summary
FLDX.S base,index,result	load indexed 32-bit value
FLDX.D base,index,result	load indexed 64-bit value
FLDX.P base,index,result	load indexed two 32-bit values
FSTX.S base,index,value	store indexed 32-bit value
FSTX.D base,index,value	store indexed 64-bit value
FSTX.P base,index,value	store indexed two 32-bit values

Table 89: Aligned floating-point load and store instructions with indexed addressing

SuperH

9 SHmedia system instructions

9.1 Overview

System instructions are used for supporting the event handling mechanism, and for accessing control and configuration registers The instructions that support event handling are described in *Section 9.2*. Instructions to access control registers are described in *Section 9.3*, and instructions to access configuration registers are described in *Section 9.4*.

9.2 Event handling instructions

There are three system instructions related to event handling. Further information on event handling can be found in *Chapter 16: Event handling on page 221*.

Instruction	Summary
RTE	return from exception
TRAPA tra	cause a trap
BRK	cause a break

Table 90: Event handling instructions

RTE is used to return from an event handler, and has no operands. RTE is a privileged instruction and raises a RESINST exception if executed in user mode. The actions performed by RTE are described in *Section 16.7: Recovery on page 235*.

TRAPA is used to cause a trap exception to be taken unconditionally. TRAPA has a register operand which is used when initializing the TRA control register during a trap handler launch. The trap exception is called TRAP and is described in *Section 16.11.2: Instruction opcode exceptions on page 243*.

BRK is used to cause a debug exception to be taken unconditionally. BRK has no operands. The debug exception is called BREAK and is described in *Section 16.11.5: Debug exceptions on page 247*. The BRK instruction is typically reserved for use by the debugger.

9.3 Control registers

The architecture defines sixty-four, 64-bit control registers held in a control register set. These provide a uniform mechanism for accessing the state used to control the CPU. Control registers implicitly affect the execution of instructions.

Many of the control registers relate to reset, interrupt, exception and panic handling, and these interactions are described in *Chapter 16: Event handling on page 221*.

9.3.1 Control register set

CR denotes the set of control registers. CR_0 to CR_{31} are privileged control registers, and CR_{32} to CR_{63} are user-accessible control registers. The behavior of each control register can be different in privileged mode or user mode. The behaviors are selected from the following list:

- DEFINED: the control register has an architecturally defined behavior for reads and writes.
- UNDEFINED: the control register has no architecturally defined behavior for reads and writes. This behavior does not occur for user mode accesses, so that user mode programs are not able to cause architecturally undefined behavior.
- EXCEPTION: reads and writes to the control register cause a reserved instruction exception. This behavior is used to detect access to privileged control registers from user mode. It does not occur for privileged mode accesses.
- RESERVED: reads to the control register return zero, and writes to the control register are ignored.

The DEFINED control registers are described in *Chapter 15: Control registers on page 207*.

Where a control register exhibits UNDEFINED, EXCEPTION or RESERVED behavior, software should not access that register. These registers could be used to extend the architecture in some future revision. Future implementations could exhibit different behavior to that described above.

9.3.2 Control register instructions

Control registers are accessed using two instructions. GETCON performs a 64-bit data transfer from a control register to a general-purpose register. PUTCON performs a 64-bit data transfer from a general-purpose register to a control register.

Instruction	Summary
GETCON index,result	move from control register
PUTCON value,index	move to control register

Table 91: Control register instructions

The behavior of GETCON and PUTCON is summarized in the following table.

Behavior	Privileged mode		User mode	
	GETCON	PUTCON	GETCON	PUTCON
DEFINED	Gets control register value	Puts control register value	Gets control register value	Puts control register value
UNDEFINED	Returns architecturally undefined value	Causes architecturally undefined behavior	(does not occur)	
RESERVED	Reads as zero	Write ignored	Reads as zero	Write ignored
EXCEPTION	(does not occur)		Reserved instruction exception is raised	

Table 92: GETCON and PUTCON behavior

GETCON and PUTCON are precise. All immediate side-effects of a PUTCON instruction take effect on the completion of the PUTCON instruction, and are visible to the immediately following instructions. In some situations, additional software actions are required to ensure correct architectural behavior. These requirements are explicitly described in the text.

9.4 Configuration registers

Configuration registers are used to configure highly implementation-dependent parts of the CPU, such as memory management and the cache. Configuration registers hold arrays of state and are accessed through a configuration register space.

An implementation with an MMU provides MMU configuration registers. An implementation with caches provides cache configuration registers. Other configuration registers can be provided depending on the implementation. The configuration register map is not defined in this document because its contents are highly implementation dependent.

9.4.1 Configuration register space

The architecture provides a configuration space containing 2^{32} 64-bit configuration registers. This space is unrelated to the memory address space. It is not translated and it is not accessible by load and store instructions. It is not byte-addressed; it is addressed by configuration register number. The notation CFG[i] refers to the $i^{th.}$ 64-bit configuration register.

The set of defined configuration registers is implementation specific. However, configuration registers with indices outside of the range $[0, 2^{32})$ are undefined on all implementations. Reading from an undefined configuration register gives an architecturally-undefined value. Writing to an undefined configuration register gives architecturally-undefined behavior.

9.4.2 Configuration register instructions

Configuration registers are accessed using two instructions. These are privileged instructions. Execution of these two instructions in user mode will result in a reserved instruction exception. GETCFG performs a 64-bit data transfer from a configuration register to a general-purpose register. PUTCFG performs a 64-bit data transfer from a general-purpose register to a configuration register. The configuration register is identified by adding a base value (provided in a register) with an offset value (provided as an immediate).

Instruction	Summary
GETCFG base,offset,result	move from configuration register
PUTCFG base,offset,value	move to configuration register

Table 93: Configuration register instructions

GETCFG and PUTCFG are precise. All immediate side-effects of a PUTCFG instruction take effect on the completion of the PUTCFG instruction, and are visible to the immediately following instructions. In some situations, additional software actions are required to ensure correct architectural behavior. These requirements are explicitly described in the text.

SHcompact instructions 10

10.1 Overview

All SHcompact instructions are 2 bytes in length, and are held in memory on 2-byte boundaries. Instructions are described as collections of 16 bits, numbered from 0 (the least significant bit) to 15 (the most significant bit). The endianness of instructions in memory is dictated by the endianness of the processor.

If the processor is little endian, instructions are held in little-endian order in memory (see *Figure 51*). The least significant byte of an instruction, containing bits 0 to 7 of its encoding, is held at the lower address in the memory representation (at address A). The most significant byte of this instruction, containing bits 8 to 15 of its encoding, is held at the higher address (at address A+1).

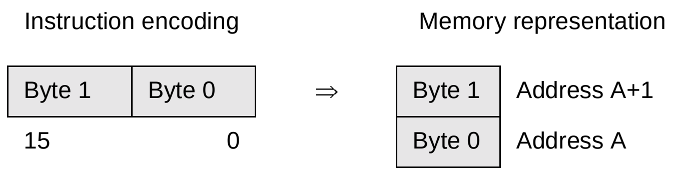


Figure 51: Little-endian memory representation of an SHcompact instruction

Alternatively, if the processor is big endian instructions are held in big-endian order in memory (see *Figure 52*). The most significant byte of an instruction, containing bits 8 to 15 of its encoding, is held at the lower address in the memory representation (at address A). The least significant byte of this instruction, containing bits 0 to 7 of its encoding, is held at the higher address (at address A+1).

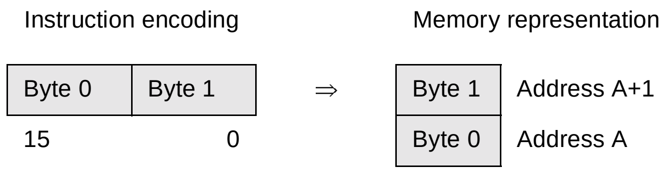


Figure 52: Big-endian memory representation of an SHcompact instruction

The following chapters (*Chapter 11 on page 171* to *Chapter 14 on page 205*) summarize the SHcompact instruction set. Further details can be found in *Volume 3 Chapter 2: SHcompact instruction set*.

10.2 Formats

Every SHcompact instruction is associated with an instruction format. The format of an instruction determines how that instruction is encoded and decoded. Each instruction format contains 16 bits, and these are grouped together into bit-fields. Each bit-field is a contiguous collection of bits and is associated with a bit-field type.

The available bit-field types are denoted by single character identifiers:

- *x*: instruction opcode. The opcode uniquely identifies an instruction.
- *m*: source register. This bit-field type identifies the right source operand.
- *n*: destination/source register. For a single-source (monadic) instruction, this bit-field type identifies the destination. For a double-source (dyadic) instruction, this bit-field type identifies the left source operand, as well as the destination operand. In the latter case, the left source value is overwritten with the result computed by the instruction.
- *i*: immediate. An immediate is a constant source value encoded directly in the instruction. Some instructions extract the immediate as an unsigned number, while others extract it as a signed number.
- *d*: displacement. A displacement is a constant source value encoded directly in the instruction. This bit-field type is used in address calculations performed by load, store and branch instructions. Some instructions extract the displacement as an unsigned number, while others extract it as a signed number.

SHcompact instruction formats are specified in *Volume 3, Appendix A: SHcompact instruction encoding*.

11 SHcompact integer instructions

11.1 Overview

This chapter describes the SHcompact integer instructions.

These instructions are classified as follows:

1 Control flow instructions: these instructions are used to determine the control flow through the program.

2 Arithmetic instructions: these instructions perform integer arithmetic. The operations include addition, subtraction, multiplication, support for division, multiply-and-accumulate, negation and decrement.

3 Comparison instructions: a set of instructions are provided for comparison of signed and unsigned integer data, and of string data.

4 No-operation: an instruction is provided to perform no operation.

5 Bitwise instructions: these instructions perform bitwise operations on integer data. The set of operations include bitwise AND, OR, XOR and NOT.

6 Rotate and shift instructions: these instructions perform rotations of integer data, arithmetic shifts and logical shifts. Shift amounts can be specified in immediates or registers.

7 Miscellaneous instructions: this class includes instructions to move values between registers, to load an immediate value into a register, and to read the T-bit. Instructions are also provided to construct PC-relative effective addresses, to swap bytes and words in registers, and to extract data from a pair of registers.

8 Special instructions: these instructions operate on special registers and special flags. The special registers are the multiply-and-accumulate low and high registers (MACL and MACH), the procedure register (PR) and the global base

register (GBR). Instructions are defined to move values between general-purpose registers and special registers, to move values between memory and special registers and to clear the multiply-and-accumulate registers. The instructions for the special flags can clear and set the S-bit and the T-bit.

11.2 Control flow instructions

These instructions determine the control flow through the program.

SHcompact supports delayed branches. The instruction immediately following a delayed branch in memory is called a delay slot instruction. The delayed branch and the delay slot are still executed in their program order. However, for a delayed branch, the delay slot is executed before the branch is effected. This means that the delay slot is executed regardless of whether the branch is taken or not.

For a non-delayed branch, there is no delay slot and the branch is effected immediately after the execution of the non-delayed branch instruction.

Conditional branches

These instructions cause a conditional transfer of control depending on the value of the T-bit. The T-bit is usually set by a previous compare instruction.

Instruction	Summary
BF label	non-delayed branch to PC-relative label address if T-bit is 0
BF/S label	delayed branch to PC-relative label address if T-bit is 0
BT label	non-delayed branch to PC-relative label address if T-bit is 1
BT/S label	delayed branch to PC-relative label address if T-bit is 1

Table 94: Conditional branch instructions

Unconditional branch

This instruction causes an unconditional transfer of control.

Instruction	Summary
BRA label	delayed branch to PC-relative label address

Table 95: Unconditional branch

Branch subroutine

This instruction causes an unconditional transfer of control, and also sets the procedure register (PR). The value of PR can be used to return to the instruction following the branch sub-routine instruction. The target instruction executes in SHcompact mode.

Instruction	Summary
BSR label	delayed branch to PC-relative label address, set the procedure register

Table 96: Branch sub-routine

Branches with mode switch

The mode of operation can be changed using the least significant bit of the target address of these instructions. A ‘0’ in this bit indicates that the target instruction will execute in SHcompact while a ‘1’ indicates SHmedia. The mode switching behavior of these instructions correspond to instruction address error exceptions on SH-4.

The least significant bit is only used for mode indication, and is masked out to give the target program counter value. All mode-switching instructions are unconditional delayed branches. The mode switch occurs after the delay slot has been executed.

Instruction	Summary
BRAF Rn	delayed branch to PC-relative target address
BSRF Rn	delayed branch to PC-relative target address, set the procedure register
JMP @Rn	delayed branch to absolute target address
JSR @Rn	delayed branch to absolute target address, set the procedure register
RTS	delayed branch to absolute target address held in the procedure register

Table 97: Mode-switching branch instructions

11.3 Arithmetic instructions

These instructions perform standard arithmetic operations on 32-bit data (except where stated otherwise). Some instructions use an immediate value for one of the operands, and some operate on data held in memory.

The DIV0S, DIV0U and DIV1 instructions use M, Q and T bits in the status register to hold additional state. The S-bit determines whether MAC.L and MAC.W calculate a saturated result.

Instruction	Summary
ADD Rm, Rn	add register to register
ADD #imm, Rn	add register to immediate
ADDC Rm, Rn	add register to register with carry
ADDV Rm, Rn	add register to register with overflow check
DIV0S Rm, Rn	divide step 0 as signed
DIV0U	divide step 0 as unsigned
DIV1 Rm, Rn	divide step 1
DMULS.L Rm, Rn	double-length multiply as signed (32 x 32 to 64)
DMULU.L Rm, Rn	double-length multiply as unsigned (32 x 32 to 64)
DT Rn	decrement register and test for zero
MAC.L @Rm+, @Rn+	multiply and accumulate long, operands from memory
MAC.W @Rm+, @Rn+	multiply and accumulate word, operands from memory
MUL.L Rm, Rn	multiply long (32 x 32 to 32)
MULS.W Rm, Rn	multiply signed word (16 x 16 to 32)
MULU.W Rm, Rn	multiply unsigned word (16 x 16 to 32)
NEG Rm, Rn	negate register
NEGC Rm, Rn	negate register with carry
SUB Rm, Rn	subtract register from register

Table 98: Arithmetic instructions

Instruction	Summary
SUBC Rm, Rn	subtract register from register with carry
SUBV Rm, Rn	subtract register from register with underflow check

Table 98: Arithmetic instructions

11.4 Comparison instructions

These instructions compare 32-bit values and place a boolean result in the T-bit.

Instruction	Summary
CMP/EQ Rm, Rn	compare register equal to register
CMP/EQ #imm, R0	compare register equal to immediate
CMP/GE Rm, Rn	compare register greater than or equal to register
CMP/GT Rm, Rn	compare register greater than register
CMP/HI Rm, Rn	compare register higher than register
CMP/HS Rm, Rn	compare register higher or same as register
CMP/PL Rn	compare register greater than 0
CMP/PZ Rn	compare register greater equal 0
CMP/STR Rm, Rn	compare string data held in registers

Table 99: Comparison instructions

11.5 No-operation

The no-operation instruction is often used to pad out the instruction text to a required alignment. It is also often used to fill empty delay slots.

Instruction	Summary
NOP	no operation

Table 100: No operation

11.6 Bitwise instructions

These instructions performing standard bitwise operations. Operations are performed on 32-bit data except where explicitly specified otherwise.

Bitwise operations

These include monadic and dyadic operations. Some instructions use an immediate value for one of the operands, and some operate on data held in memory.

Instruction	Summary
AND Rm, Rn	bitwise AND of register with register
AND #imm, R0	bitwise AND of register with immediate value
AND.B #imm, @(R0, GBR)	bitwise AND of immediate with 8-bit memory value
NOT Rm, Rn	bitwise NOT of register
OR Rm, Rn	bitwise OR of register with register
OR #imm, R0	bitwise OR of register with immediate value
OR.B #imm, @(R0, GBR)	bitwise OR of immediate with 8-bit memory value
XOR Rm, Rn	bitwise XOR of register with register
XOR #imm, R0	bitwise XOR of register with immediate value
XOR.B #imm, @(R0, GBR)	bitwise XOR of immediate with 8-bit memory value

Table 101: Bitwise instructions

Test operations

These instructions perform tests on data held in a register or in memory. The result is placed in the T-bit.

Instruction	Summary
TST Rm, Rn	bitwise AND of register with register to set T-bit
TST #imm, R0	bitwise AND of register with immediate value to set T-bit
TST.B #imm, @(R0, GBR)	bitwise AND of immediate with 8-bit memory value to set T-bit

Table 102: Test instructions

11.7 Rotate and shift instructions

These instructions perform rotate and shifts on a register operand.

Rotates

Instruction	Summary
ROTCL Rn	rotate left T-bit and register
ROTCR Rn	rotate right T-bit and register
ROTL Rn	rotate left register, T-bit gets most significant bit
ROTR Rn	rotate right register, T-bit gets least significant bit

Table 103: Rotate instructions

Shifts

Instruction	Summary
SHAD Rm, Rn	dynamic arithmetic shift of register
SHAL Rn	1-bit left shift of register
SHAR Rn	1-bit arithmetic right shift of register
SHLD Rm, Rn	dynamic logical shift of register
SHLL Rn	shift logical left by 1
SHLL2 Rn	shift logical left by 2
SHLL8 Rn	shift logical left by 8
SHLL16 Rn	shift logical left by 16
SHLR Rn	shift logical right by 1
SHLR2 Rn	shift logical right by 2
SHLR8 Rn	shift logical right by 8
SHLR16 Rn	shift logical right by 16

Table 104: Shift instructions

11.8 Miscellaneous instructions

These instructions perform various moves into registers, and perform register to register transfers with some data reformatting. Move instructions that access memory are described in *Chapter 12: SHcompact memory instructions on page 181*.

Various Move Instructions

Instruction	Summary
MOV #imm, Rn	move immediate value to a register
MOV Rm, Rn	register to register move, copies 64-bit data
MOVA @(disp, PC), R0	calculate a PC-relative effective address
MOVT Rn	move T-bit to a register

Table 105: Various move instructions

Register to register transfers

Instruction	Summary
EXTS.B Rm, Rn	sign-extend 8-bit data in register
EXTS.W Rm, Rn	sign-extend 16-bit data in register
EXTU.B Rm, Rn	zero-extend 8-bit data in register
EXTU.W Rm, Rn	zero-extend 16-bit data in register
SWAP.B Rm, Rn	swap register bytes
SWAP.W Rm, Rn	swap register words
XTRCT Rm, Rn	extract a long-word from a pair of registers

Table 106: Register transfer instructions

11.9 Special instructions

These instructions are used for manipulating special state.

Loading a special register

Instruction	Summary
LDC Rm, GBR	load from register to GBR
LDC.L @Rm+, GBR	load from memory to GBR with post-increment
LDS Rm, MACH	load from register to MACH
LDS.L @Rm+, MACH	load from memory to MACH with post-increment
LDS Rm, MACL	load from register to MACL
LDS.L @Rm+, MACL	load from memory to MACL with post-increment
LDS Rm, PR	load from register to PR
LDS.L @Rm+, PR	load from memory to PR with post-increment

Table 107: Load special register instructions

Storing a special register

Instruction	Summary
STC GBR, Rn	store to register from GBR
STC.L GBR, @-Rn	store to memory from GBR with pre-decrement
STS MACH, Rn	store to register from MACH
STS.L MACH, @-Rn	store to memory from MACH with pre-decrement
STS MACL, Rn	store to register from MACL
STS.L MACL, @-Rn	store to memory from MACL with pre-decrement
STS PR, Rn	store to register from PR
STS.L PR, @-Rn	store to memory from PR with pre-decrement

Table 108: Store special register instructions

Clearing and setting special registers and flags

Instruction	Summary
CLRMAC	clear MACL and MACH registers
CLRS	clear S-bit
CLRT	clear T-bit
SETS	set S-bit
SETT	set T-bit

Table 109: Clear special register and flag instructions

12 SHcompact memory instructions

This chapter defines the SHcompact load, store and cache instructions for general-purpose registers. The load/store instructions for floating-point registers are described separately in *Chapter 13: SHcompact floating-point on page 189*.

The endianness of the data is properly converted when loaded from memory to a register or when stored from a register into memory.

The following datatypes are supported for load/store instructions:

- Byte (suffix ‘.B’): 8 bits of data.
- Word (suffix ‘.W’): 16 bits of data.
- Long-word (suffix ‘.L’): 32 bits of data.

All byte and word data is sign-extended when loaded. Load and store instructions raise an exception if the accessed data is not naturally aligned in memory. The effective address for an access of width n bytes should be an exact multiple of n, otherwise an exception is raised to indicate misalignment.

12.1 Load/store instructions

These instructions are used to move data between general-purpose registers and memory. The instructions are grouped here according to their addressing mode.

PC indirect with displacement

The effective address is calculated by adding a displacement to PC. This addressing mode is only available for word and long-word loads.

Instruction	Summary
MOV.W @(disp, PC), Rn	load 16-bits indirect from PC with displacement
MOV.L @(disp, PC), Rn	load 32-bits indirect from PC with displacement

Table 110: Memory instructions using PC indirect with displacement addressing

GBR indirect with displacement

The effective address is calculated by adding a displacement to GBR.The displacement is scaled by the width of the access.

Instruction	Summary
MOV.B R0, @(disp, GBR)	store 8-bits indirect to GBR with displacement
MOV.W R0, @(disp, GBR)	store 16-bits indirect to GBR with displacement
MOV.L R0, @(disp, GBR)	store 32-bits indirect to GBR with displacement
MOV.B @(disp, GBR), R0	load 8-bits indirect from GBR with displacement
MOV.W @(disp, GBR), R0	load 16-bits indirect from GBR with displacement
MOV.L @(disp, GBR), R0	load 32-bits indirect from GBR with displacement

Table 111: Memory instructions using GBR indirect with displacement addressing

Register indirect

The effective address is specified in a register.

Instruction	Summary
MOV.B Rm, @Rn	store 8-bits indirect
MOV.W Rm, @Rn	store 16-bits indirect
MOV.L Rm, @Rn	store 32-bits indirect
MOV.B @Rm, Rn	load 8-bits indirect
MOV.W @Rm, Rn	load 16-bits indirect
MOV.L @Rm, Rn	load 32-bits indirect

Table 112: Memory instructions using register indirect addressing

Register indirect with displacement

The effective address is calculated by adding a register and a displacement. The displacement is scaled by the width of the access.

Instruction	Summary
MOV.B R0, @(disp, Rn)	store 8-bits indirect with displacement
MOV.W R0, @(disp, Rn)	store 16-bits indirect with displacement
MOV.L Rm, @(disp, Rn)	store 32-bits indirect with displacement
MOV.B @(disp, Rm), R0	load 8-bits indirect with displacement
MOV.W @(disp, Rm), R0	load 16-bits indirect with displacement
MOV.L @(disp, Rm), Rn	load 32-bits indirect with displacement

Table 113: Memory instructions using register indirect with displacement addressing

Register indirect with pre-decrement

A register is pre-decremented by the width of the access, and the resulting value specifies the effective address. This addressing mode is only available for stores.

Instruction	Summary
MOV.B Rm, @-Rn	store 8-bits indirect with pre-decrement
MOV.W Rm, @-Rn	store 16-bits indirect with pre-decrement
MOV.L Rm, @-Rn	store 32-bits indirect with pre-decrement

Table 114: Memory instructions using register indirect with pre-decrement addressing

Register indirect with post-increment

The effective address is specified in a register. This register is post-incremented by the width of the access. This addressing mode is only available for loads.

Instruction	Summary
MOV.B @Rm+, Rn	load 8-bits indirect with post-increment
MOV.W @Rm+, Rn	load 16-bits indirect with post-increment
MOV.L @Rm+, Rn	load 32-bits indirect with post-increment

Table 115: Memory instructions using register indirect with post-increment addressing

Register indirect with indexing

The effective address is calculated by adding two registers together.

Instruction	Summary
MOV.B Rm, @(R0, Rn)	store 8-bits indirect with indexing
MOV.W Rm, @(R0, Rn)	store 16-bits indirect with indexing
MOV.L Rm, @(R0, Rn)	store 32-bits indirect with indexing
MOV.B @(R0, Rm), Rn	load 8-bits indirect with indexing
MOV.W @(R0, Rm), Rn	load 16-bits indirect with indexing
MOV.L @(R0, Rm), Rn	load 32-bits indirect with indexing

Table 116: Memory instructions using register indirect with indexing addressing

12.2 Test and set instruction

This instruction performs a test-and-set operation on the byte data at the effective address specified in R_n. The 8 bits of data at the effective address are read from memory. If the read data is 0 the T-bit is set, otherwise the T-bit is cleared. The highest bit of the 8-bit data (bit 7) is then set, and the result is written back to the memory at the same effective address.

This test-and-set is atomic from the CPU perspective. This instruction cannot be interrupted during its operation. However, atomicity is not provided with respect to accesses from other memory users. It is possible that a memory access from another memory user could observe or modify the memory state between the read and write.

Instruction	Summary
TAS.B @Rn	test 8-bit memory value and set T-bit

Table 117: Test and set instruction

There is no special treatment for TAS.B regarding the cache, and it behaves in the same way as a load followed by a store. Depending on the cache behavior, it is possible for the TAS.B accesses to be completed in the cache with no external memory activity. If the MMU is disabled, the accesses will be performed on external memory. If the MMU is enabled, the accesses will be performed according to the cache behavior of the translation:

- For device or uncached cache behavior, the access will be performed on external memory locations.
- For write-through and write-back cache behaviors, the read can be performed on the cache if there is a cache hit. If there is a cache miss the read access can cause a cache block to be allocated and this can cause other cache blocks to be evicted or written-back from the cache. For write-through cache behavior, the write will be written to external memory. For write-back cache behavior the write can be completed in the cache (if it hits) without an external memory write. Note that the full behavior depends on other details of cache operation.

As described earlier, the read and write memory accesses caused by TAS.B (if any) are not guaranteed to implemented atomically on the external memory location with respect to other memory users.

The SHmedia SWAP.Q instruction (see *Section 6.5.1: Atomic swap on page 98*) provides an atomic read-modify-write on external memory, and should be used for synchronization with other memory users.

12.3 Synchronization

No SHcompact mechanisms are provided for:

- Data synchronization
- Instruction synchronization
- Memory synchronization that is atomic with respect to other memory users.

The SHmedia mechanisms described in *Section 6.5: Synchronization on page 98* should be used instead.

12.4 Cache instructions

There are 3 categories of cache instruction: prefetch (PREF), allocate (MOVCA.L) and coherency (OCBI, OCBP and OCBWB) instructions.

These instructions allow software to control and optimize cache operation in a largely implementation-independent manner. Note that the cache block size is exposed by these instructions and this value is implementation specific.

Instruction	Summary
MOVCA.L R0, @Rn	move with cache block allocation
OCBI @Rn	operand cache block invalidate
OCBP @Rn	operand cache block purge
OCBWB @Rn	operand cache block writeback
PREF @Rn	prefetch data from memory to operand cache

Table 118: Cache instructions

These instructions have behaviors corresponding to SHmedia instructions described in *Section 6.6: Cache instructions on page 101*.

SHcompact Instruction	Analogous SHmedia Instructions
MOVCA.L R0, @Rn	ALLOCO of address in Rn, followed by a 32-bit store of R0 to address in Rn
OCBI @Rn	OCBI of address in Rn
OCBP @Rn	OCBP of address in Rn
OCBWB @Rn	OCBWB of address in Rn
PREF @Rn	Aligned load from address in Rn to R63 (data prefetch)

Table 119: Cache instructions

SHcompact does not provide instructions for invalidation or prefetch of instruction cache blocks. The SHmedia mechanisms described in *Section 6.6: Cache instructions on page 101* should be used instead.

Further information on caches can be found in *Chapter 18: Caches on page 297*.

SuperH

SHcompact floating-point 13

13.1 Overview

SHcompact provides a comprehensive set of floating-point instructions for single-precision and double-precision representations. The SHcompact floating-point instructions are the set of instructions that are defined in this chapter. The SHcompact floating-point state consists of the floating-point register set, FPSCR and FPUL. These are defined in *Section 2.9: SHcompact state on page 26*.

The IEEE754 floating-point standard is supported through a combination of hardware and system software. This is described in *Section 8.3: IEEE754 floating-point support on page 136*.

The architecture also provides non-IEEE754 support including fast handling of denormalized numbers, fused multiply accumulate and special-purpose instructions. This is described in *Section 8.4: Non-IEEE754 floating-point support on page 145*.

13.2 Floating-point disable

The architecture allows the floating-point unit to be disabled by software. This is achieved by setting SR.FD to 1. Once disabled, any attempt to execute a floating-point opcode will generate an exception. The set of floating-point opcodes is described in *Volume 3, Appendix A: SHcompact instruction encoding*.

If an implementation does not provide a floating-point unit, then the behavior of floating-point instructions is the same as an implementation with a floating-point unit that is permanently disabled. On such an implementation, SR.FD is permanently set to 1 and the implementation does not provide the floating-point state. Any attempt to access the floating-point state will generate an exception. It is possible to emulate the floating-point instructions and state in software.

13.3 Floating-point register set

The SHcompact floating-point register set is described in *Chapter 2: Architectural state on page 13*. In SHcompact mode, it consists of two banks, each containing 16 single-precision (32 bit) floating-point registers. These registers can also be viewed as double-precision (64 bit) values, pairs of single-precision values, 4-element vectors of single-precision values and 16-element matrices of single-precision values.

13.4 FPSCR

The floating-point status and control register, FPSCR, is 32 bits wide. It is used to control, read and write the floating-point status. The SHcompact view of FPSCR is shown in *Figure 53*. The ‘r’ field indicates reserved bits.

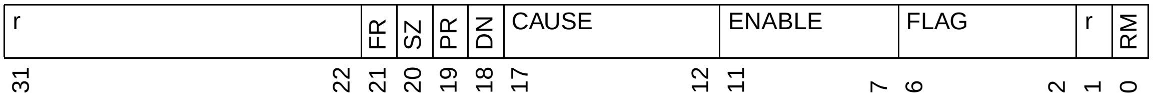


Figure 53: FPSCR

The RM, FLAG, ENABLE, CAUSE and DN fields have the behavior described in *Section 8.6.1: Floating-point status and control on page 148*.

The SHmedia view of FPSCR does not contain the PR, SZ and FR fields. The mapping between the SHmedia and SHcompact views are described in *Section 2.9.2: SHcompact floating-point register state on page 27*.

These 3 fields are used to provide additional encoding information for SHcompact floating-point instructions. They are used as follows:

- FPSCR.PR selects the precision of operation: 0 indicates single-precision and 1 indicates double-precision. Some floating-point instructions are only available when FPSCR.PR has a certain value.
- FPSCR.SZ selects the width of data-transfer for floating-point loads and stores: 0 indicates transfers of 32-bit registers and 1 indicates transfers of pairs of

32-bit registers (64 bits). Some floating-point instructions are only available when FPSCR.SZ has a certain value.

- FPSCR.FR selects which bank is viewed using the regular floating-point register names and which as the extended bank: the banking arrangement is described in *Section 2.9.2: SHcompact floating-point register state on page 27*.

13.5 FPUL

FPUL is a 32-bit register used to move and convert data between general-purpose registers and floating-point registers. All operations using FPUL transfer 32 bits of data into or out of FPUL. This can be a long-word integer or a single-precision floating-point value.

The operations that use FPUL are listed in *Table 120* and *Table 121*.

Operation	Source	Destination
Move	FPUL	Single-precision floating-point register
Move	FPUL	General-purpose register
Move	FPUL	Memory
Convert long-word to single	FPUL	Single-precision floating-point register
Convert long-word to double	FPUL	Double-precision floating-point register
Convert single to double	FPUL	Double-precision floating-point register

Table 120: Operations that read from FPUL

Operation	Source	Destination
Move	Single-precision floating-point register	FPUL
Move	General-purpose register	FPUL
Move	Memory	FPUL
Convert single to long-word	Single-precision floating-point register	FPUL
Convert double to long-word	Double-precision floating-point register	FPUL
Convert double to single	Double-precision floating-point register	FPUL

Table 121: Operations that write to FPUL

13.6 Floating-point instructions

This section describes the SHcompact floating-point instructions. The SHcompact floating-point provision is largely analogous to that for SHmedia. For further details, refer to *Chapter 8: SHmedia floating-point on page 135*.

13.6.1 Floating-point special register access

These instructions support access to the FPSCR and FPUL registers. All of these accesses transfer 32 bits of data.

Instruction	Summary
FLDS FRm, FPUL	floating-point load from register to FPUL
FSTS FPUL, FRn	floating-point store to register from FPUL
LDS Rm, FPSCR	load from register to FPSCR
LDS.L @Rm+, FPSCR	load from memory to FPSCR with post-increment
LDS Rm, FPUL	load from register to FPUL
LDS.L @Rm+, FPUL	load from memory to FPUL with post-increment
STS FPSCR, Rn	store to register from FPSCR
STS.L FPSCR, @-Rn	store to memory from FPSCR with pre-decrement
STS FPUL, Rn	store to register from FPUL
STS.L FPUL, @-Rn	store to memory from FPUL with pre-decrement

Table 122: Floating-point special register access instructions

13.6.2 Floating-point constant loading

These instructions are used to load single-precision constants 0.0 and 1.0 into floating-point registers:

Instruction	Summary
FLDI0 FRn	single floating-point load of 0.0
FLDI1 FRn	single floating-point load of 1.0

Table 123: Floating-point constant loading instructions

13.6.3 Floating-point dyadic arithmetic

This set of floating-point instructions operate on two floating-point source operands. The following instructions are defined:

Instruction	Summary
FADD DRm, DRn	double floating-point add
FADD FRm, FRn	single floating-point add
FDIV DRm, DRn	double floating-point divide
FDIV FRm, FRn	single floating-point divide
FMUL DRm, DRn	double floating-point multiply
FMUL FRm, FRn	single floating-point multiply
FSUB DRm, DRn	double floating-point subtract
FSUB FRm, FRn	single floating-point subtract

Table 124: Floating-point dyadic arithmetic instructions

13.6.4 Floating-point monadic arithmetic

This set of floating-point instructions operate on one floating-point source operand. The following instructions are defined:

Instruction	Summary
FABS DRn	double floating-point absolute
FABS FRn	single floating-point absolute
FNEG DRn	double floating-point negate
FNEG FRn	single floating-point negate
FSQRT DRn	double floating-point square root
FSQRT FRn	single floating-point square root

Table 125: Floating-point monadic arithmetic instructions

13.6.5 Floating-point multiply and accumulate

A floating-point multiply and accumulate instruction is defined to multiply two source operands and then add this to a third source operand. The third source operand also defines the destination into which the result is placed. The effect is that the output of the multiplication is accumulated into the result.

Instruction	Summary
FMAC FR0, FRm, FRn	single floating-point multiply and accumulate

Table 126: Floating-point multiply and accumulate

In terms of register operands, the FMAC instruction computes $(FR_0 \times FR_m) + FR_n$ and places the result in FR_n.

13.6.6 Floating-point comparisons

The following comparison instructions are defined:

Instruction	Summary
FCMP/EQ DRm, DRn	double floating-point compare equal
FCMP/EQ FRm, FRn	single floating-point compare equal
FCMP/GT DRm, DRn	double floating-point compare greater
FCMP/GT FRm, FRn	single floating-point compare greater

Table 127: Floating-point comparison instructions

The boolean result of the comparison is placed in the T-bit.

13.6.7 Floating-point conversions

A set of conversion instructions are defined to convert between the two floating-point formats, and between integer and floating-point formats:

Instruction	Summary
FCNVDS DRm, FPUL	double to single floating-point convert
FCNVSD FPUL, DRn	single to double floating-point convert
FLOAT FPUL, DRn	double floating-point convert from integer
FLOAT FPUL, FRn	single floating-point convert from integer
FTRC DRm, FPUL	double floating-point truncate and convert to integer
FTRC FRm, FPUL	single floating-point truncate and convert to integer

Table 128: Floating-point conversion instructions

13.6.8 Special-purpose floating-point instructions

These instructions are mainly used in special-purpose applications (for example, graphics). The special-purpose floating-point instructions are:

Instruction	Summary
FIPR FVm, FVn	single floating-point inner product
FTRV XMTRX, FVn	single floating-point transform vector
FSCA FPUL, DRn	single floating-point sine cosine approximate
FSRRA FRn	single reciprocal square root approximate

Table 129: Special-purpose floating-point instructions

These instructions do not conform to the IEEE754 standard and return approximate results. An implementation will return a result which is strictly within a specified error bound relative to the fully precise result. The numerical properties of each of these SHcompact instructions is the same as the corresponding SHmedia instruction. Further information can be found in *Section 8.7: Special-purpose floating-point instructions on page 154*.

FIPR

The numerical properties of the SHcompact FIPR instruction correspond to those of the SHmedia FIPR.S instruction. The correspondence between the FIPR and FIPR.S operands is as follows:

- FV_m source operand of FIPR corresponds to the FV_h operand of FIPR.S
- FV_n source operand of FIPR corresponds to the FV_g operand of FIPR.S
- FR_{n+3} result operand of FIPR corresponds to the FR_f operand of FIPR.S

FTRV

Similarly, the numerical properties of the SHcompact FTRV instruction correspond to those of the SHmedia FTRV.S instruction. The correspondence between the FTRV and FTRV.S operands is as follows:

- XMTRX source operand of FTRV corresponds to the $MTRX_g$ operand of FTRV.S
- FV_n source operand of FTRV corresponds to the FV_h operand of FTRV.S
- FV_n result operand of FTRV corresponds to the FV_f result operand of FTRV.S

FSCA

FSCA produces two single-precision results, representing the sine and the cosine of the source operand. These two results are placed into a pair of single-precision registers which are denoted in *Table 129* using the double-precision notation DR_n. The most significant half of DR_n (also known as FR_n) contains the sine result, and the least significant half of DR_n (also known as FR_{n+1}) contains the cosine result.

The numerical properties of FSCA correspond to those of the SHmedia FSINA.S and FCOSA.S instructions.

FSRRA

FSRRA produces a single-precision result which is an approximation to the reciprocal square-root of the source operand. The numerical properties of FSRRA correspond to those of the SHmedia FSRRA.S instruction.

13.6.9 Floating-point width and bank change

These instructions are used to toggle the state of the FPSCR.FR and FPSCR.SZ bits: No analogous instruction is provided to toggle the state of the FPSCR.PR bit. These bits are described in *Section 13.4: FPSCR on page 190*.

Instruction	Summary
FRCHG	FR-bit change
FSCHG	SZ-bit change

Table 130: Floating-point width and bank change instructions

13.6.10 Floating-point move instructions

These instructions are used to transfer values between floating-point registers:

Instruction	Summary
FMOV FRm, FRn	single to single floating-point move
FMOV DRm, DRn	single-pair to single-pair floating-point move
FMOV DRm, XDn	single-pair to extended single-pair floating-point move
FMOV XDm, DRn	extended single-pair to single-pair floating-point move
FMOV XDm, XDn	extended single-pair to extended single-pair floating-point move

Table 131: Floating-point move instructions

FMOV FRm, FRn is a single-precision transfer and moves 32 bits of data.

The other 4 FMOV instructions in *Table 131* are double-precision transfers and move 64 bits of data. Note that a double-precision transfer actually moves a pair of single-precision registers.

13.6.11 Floating-point load/store instructions

These instructions are used to move data between floating-point registers and memory.

Load and store instructions raise an exception if the accessed data is not naturally aligned in memory. The effective address for an access of width n bytes should be an exact multiple of n, otherwise an exception is raised to indicate misalignment.

Note that a double-precision memory access is actually an access to a pair of single-precision values. These two interpretations have different behavior due to the endianness effects discussed in *Section 3.7: Data representation in memory on page 43*.

Register indirect

The effective address is specified in a general-purpose register.

Instruction	Summary
FMOV DRm, @Rn	single-pair floating-point store indirect
FMOV.S FRm, @Rn	single floating-point store indirect
FMOV XDm, @Rn	extended single-pair floating-point store indirect
FMOV @Rm, DRn	single-pair floating-point load indirect
FMOV.S @Rm, FRn	single floating-point load indirect
FMOV @Rm, XDn	extended single-pair floating-point load indirect

Table 132: Floating-point memory instructions using register indirect addressing

Register indirect with pre-decrement

A general-purpose register is pre-decremented by the width of the access, and the resulting value specifies the effective address. This addressing mode is only available for stores.

Instruction	Summary
FMOV DRm, @-Rn	single-pair floating-point store indirect with pre-decrement
FMOV.S FRm, @-Rn	single floating-point store indirect with pre-decrement

Table 133: Floating-point memory instructions using register indirect with pre-decrement addressing

Instruction	Summary
FMOV XDm, @-Rn	extended single-pair floating-point store indirect with pre-decrement

Table 133: Floating-point memory instructions using register indirect with pre-decrement addressing

Register indirect with post-increment

The effective address is specified in a general-purpose register. This register is post-incremented by the width of the access. This addressing mode is only available for loads.

Instruction	Summary
FMOV @Rm+, DRn	single-pair floating-point load indirect with post-increment
FMOV.S @Rm+, FRn	single floating-point load indirect with post-increment
FMOV @Rm+, XDn	extended single-pair floating-point load indirect with post-increment

Table 134: Floating-point memory instructions using register indirect with post-increment addressing

Register indirect with indexing

The effective address is calculated by adding two general-purpose registers together.

Instruction	Summary
FMOV DRm, @(R0, Rn)	single-pair floating-point store indirect with indexing
FMOV.S FRm, @(R0, Rn)	single floating-point store indirect with indexing
FMOV XDm, @(R0, Rn)	extended single-pair floating-point store indirect with indexing
FMOV @(R0, Rm), DRn	single-pair floating-point load indirect with indexing
FMOV.S @(R0, Rm), FRn	single floating-point load indirect with indexing
FMOV @(R0, Rm), XDn	extended single-pair floating-point load indirect with indexing

Table 135: Floating-point memory instructions using register indirect with indexing addressing

13.7 Reserved floating-point behavior

The behavior of SHcompact floating-point instructions is architecturally undefined when both FPSCR.PR=1 and FPSCR.SZ=1. This floating-point mode setting is reserved. Software must not rely on the behavior of SHcompact floating-point instructions in this mode, otherwise compatibility with other implementations will be impaired.

The settings for FPSCR.PR and FPSCR.SZ are summarized in *Table 136*.

FPSCR.PR	FPSCR.SZ	Behavior
0	0	Single-precision arithmetic, 32-bit load/store
0	1	Single-precision arithmetic, 2 x 32-bit load/store (pairs of singles)
1	0	Double-precision arithmetic, 32-bit load/store
1	1	Reserved, behavior is architecturally undefined

Table 136: Summary of FPSCR.PR and FPSCR.SZ settings

Even excluding the mode where FPSCR.PR=1 and FPSCR.SZ, not all SHcompact FPU instructions are available in each of the remaining 3 modes. The supported combinations are indicated by an entry of '✔' in *Table 137*. An entry of '✕' indicates that the combination is not possible due to inherent properties of the encoding. All other combinations are reserved and lead to architecturally undefined behavior when used.

Instruction	FPSCR.PR=0		FPSCR.PR=1	
	FPSCR.SZ=0	FPSCR.SZ=1	FPSCR.SZ=0	FPSCR.SZ=1
FABS DRn	✕	✕	✔	UNDEFINED
FABS FRn	✔	✔	✕	✕
FADD DRm, DRn	✕	✕	✔	UNDEFINED
FADD FRm, FRn	✔	✔	✕	✕
FCMP/EQ DRm, DRn	✕	✕	✔	UNDEFINED

Table 137: FPU instruction availability by FPSCR.PR and FPSCR.SZ settings

Instruction	FPSCR.PR=0		FPSCR.PR=1	
	FPSCR.SZ=0	FPSCR.SZ=1	FPSCR.SZ=0	FPSCR.SZ=1
FCMP/EQ FRm, FRn	✔	✔	×	×
FCMP/GT DRm, DRn	×	×	✔	UNDEFINED
FCMP/GT FRm, FRn	✔	✔	×	×
FCNVDS DRm, FPUL	UNDEFINED	UNDEFINED	✔	UNDEFINED
FCNVSD FPUL, DRn	UNDEFINED	UNDEFINED	✔	UNDEFINED
FDIV DRm, DRn	×	×	✔	UNDEFINED
FDIV FRm, FRn	✔	✔	×	×
FIPR FVm, FVn	✔	✔	UNDEFINED	UNDEFINED
FLDS FRm, FPUL	✔	✔	✔	UNDEFINED
FLDI0 FRn	✔	✔	UNDEFINED	UNDEFINED
FLDI1 FRn	✔	✔	UNDEFINED	UNDEFINED
FLOAT FPUL, DRn	×	×	✔	UNDEFINED
FLOAT FPUL, FRn	✔	✔	×	×
FMAC FR0, FRm, FRn	✔	✔	UNDEFINED	UNDEFINED
FMOV DRm, DRn	×	✔	×	UNDEFINED
FMOV DRm, XDn	×	✔	×	UNDEFINED
FMOV DRm, @Rn	×	✔	×	UNDEFINED
FMOV DRm, @-Rn	×	✔	×	UNDEFINED
FMOV DRm, @(R0, Rn)	×	✔	×	UNDEFINED
FMOV FRm, FRn	✔	×	✔	×
FMOV.S FRm, @Rn	✔	×	✔	×
FMOV.S FRm, @-Rn	✔	×	✔	×
FMOV.S FRm, @(R0, Rn)	✔	×	✔	×

Table 137: FPU instruction availability by FPSCR.PR and FPSCR.SZ settings

Instruction	FPSCR.PR=0		FPSCR.PR=1	
	FPSCR.SZ=0	FPSCR.SZ=1	FPSCR.SZ=0	FPSCR.SZ=1
FMOV XDm, DRn	×	✔	×	UNDEFINED
FMOV XDm, XDn	×	✔	×	UNDEFINED
FMOV XDm, @Rn	×	✔	×	UNDEFINED
FMOV XDm, @-Rn	×	✔	×	UNDEFINED
FMOV XDm, @(R0, Rn)	×	✔	×	UNDEFINED
FMOV @Rm, DRn	×	✔	×	UNDEFINED
FMOV @Rm+, DRn	×	✔	×	UNDEFINED
FMOV @(R0, Rm), DRn	×	✔	×	UNDEFINED
FMOV.S @Rm, FRn	✔	×	✔	×
FMOV.S @Rm+, FRn	✔	×	✔	×
FMOV.S @(R0, Rm), FRn	✔	×	✔	×
FMOV @Rm, XDn	×	✔	×	UNDEFINED
FMOV @Rm+, XDn	×	✔	×	UNDEFINED
FMOV @(R0, Rm), XDn	×	✔	×	UNDEFINED
FMUL DRm, DRn	×	×	✔	UNDEFINED
FMUL FRm, FRn	✔	✔	×	×
FNEG DRn	×	×	✔	UNDEFINED
FNEG FRn	✔	✔	×	×
FRCHG	✔	✔	UNDEFINED	UNDEFINED
FSCA FPUL, DRn	✔	✔	UNDEFINED	UNDEFINED
FSCHG	✔	✔	UNDEFINED	UNDEFINED
FSQRT DRn	×	×	✔	UNDEFINED
FSQRT FRn	✔	✔	×	×

Table 137: FPU instruction availability by FPSCR.PR and FPSCR.SZ settings

Instruction	FPSCR.PR=0		FPSCR.PR=1	
	FPSCR.SZ=0	FPSCR.SZ=1	FPSCR.SZ=0	FPSCR.SZ=1
FSRRA FRn	✔	✔	UNDEFINED	UNDEFINED
FSTS FPUL, FRn	✔	✔	✔	UNDEFINED
FSUB DRm, DRn	×	×	✔	UNDEFINED
FSUB FRm, FRn	✔	✔	×	×
FTRC DRm, FPUL	×	×	✔	UNDEFINED
FTRC FRm, FPUL	✔	✔	×	×
FTRV XMTRX, FVn	✔	✔	UNDEFINED	UNDEFINED
LDS Rm, FPSCR	✔	✔	✔	UNDEFINED
LDS.L @Rm+, FPSCR	✔	✔	✔	UNDEFINED
LDS Rm, FPUL	✔	✔	✔	UNDEFINED
LDS.L @Rm+, FPUL	✔	✔	✔	UNDEFINED
STS FPSCR, Rn	✔	✔	✔	UNDEFINED
STS.L FPSCR, @-Rn	✔	✔	✔	UNDEFINED
STS FPUL, Rn	✔	✔	✔	UNDEFINED
STS.L FPUL, @-Rn	✔	✔	✔	UNDEFINED

Table 137: FPU instruction availability by FPSCR.PR and FPSCR.SZ settings

Some SHcompact FPU instructions have reserved operand bits. These arise due to:

- Operands for double-precision and single-pair data: the lowest bit is reserved since all double-precision registers have even numbers.
- Operands for 4-element vector data: the lowest 2 bits are reserved since all 4-element vector registers have numbers that are a multiple of 4.

In all cases the required settings for any reserved operand bits are fully specified in the architecture description of the SHcompact encodings. The required settings vary from case to case, and the bits need to be set to 0 or 1 exactly as specified in the encodings in the instruction descriptions. In some cases, these bits are used for opcode information to distinguish different instructions. The behavior is architecturally undefined if incorrect settings are used for these bits.

Software must not rely on the behavior for incorrect settings of reserved operand bits for SHcompact floating-point instructions, otherwise compatibility with other implementations will be impaired.

SuperH

14 SHcompact system instructions

14.1 System instructions

This chapter describes the system instructions. These two instructions are related to event handling. All provided SHcompact instructions can be executed in both user and privileged modes. There are no privileged-only SHcompact instructions.

Break

BRK is used to cause a debug exception to be taken unconditionally. BRK has no operands. The debug exception is called BREAK and is described in *Section 16.11.5: Debug exceptions on page 247*. The BRK instruction is typically reserved for use by the debugger

Instruction	Summary
BRK	cause a pre-execution break exception

Table 138: Break instruction

Trap

TRAPA is used to cause a trap exception to be taken unconditionally. TRAPA has an immediate operand which is used when initializing the TRA control register during a trap handler launch. The trap exception is called TRAP and is described in *Section 16.11.2: Instruction opcode exceptions on page 243*.

Instruction	Summary
TRAPA #imm	trap always

Table 139: Trap instruction

15 Control registers

15.1 Control register set

The available control registers are largely implementation independent, though the layout of control registers has some implementation-specific properties. In particular, the width of the implemented parts of some control registers depends upon the number of bits of effective address (neff) supported by the implementation.

Table 140 and *Table 141* summarize the control register set.

Register number	Register name	Description	Behavior in privileged mode
0	SR	Status register	DEFINED (see *Section 15.2.1*)
1	SSR	Saved status register	DEFINED (see *Section 15.2.2*)
2	PSSR	Panic-saved status register	DEFINED (see *Section 15.2.3*)
3	-	Undefined control register	UNDEFINED
4	INTEVT	Interrupt event register	DEFINED (see *Section 15.2.4*)
5	EXPEVT	Exception event register	DEFINED (see *Section 15.2.4*)
6	PEXPEVT	Panic-saved exception event register	DEFINED (see *Section 15.2.4*)
7	TRA	TRAP exception register	DEFINED (see *Section 15.2.4*)
8	SPC	Saved program counter	DEFINED (see *Section 15.2.5*)
9	PSPC	Panic-saved program counter	DEFINED (see *Section 15.2.5*)

Table 140: Control register set (privileged mode)

Register number	Register name	Description	Behavior in privileged mode
10	RESVEC	Reset vector	DEFINED (see *Section 15.2.6*)
11	VBR	Vector base register	DEFINED (see *Section 15.2.7*)
12	-	Undefined control register	UNDEFINED
13	TEA	Faulting effective address register	DEFINED (see *Section 15.2.8*)
[14,15]	-	Undefined control registers	UNDEFINED
16	DCR	Debug control register	DEFINED (see *Section 15.2.9*)
17	KCR0	Kernel register 0	DEFINED (see *Section 15.2.9*)
18	KCR1	Kernel register 1	DEFINED (see *Section 15.2.9*)
[19,31]	-	Undefined control registers	UNDEFINED
[32,61]	-	Reserved control registers	RESERVED
62	CTC	Clock tick counter	DEFINED (see *Section 15.2.10*)
63	USR	User-accessible status register	DEFINED (see *Section 15.2.11*)

Table 140: Control register set (privileged mode)

Register number	Register name	Description	Behavior in user mode
[0,31]	-	Privileged-only control registers	EXCEPTION
[32,61]	-	Reserved control registers	RESERVED
62	CTC	Clock tick counter	DEFINED (see *Section 15.2.10*)
63	USR	User-accessible status register	DEFINED (see *Section 15.2.11*)

Table 141: Control register set (user mode)

DEFINED, UNDEFINED, EXCEPTION and RESERVED are defined in *Section 9.3.1: Control register set on page 164*.

15.2 Control register descriptions

The following sections describe the implementation-independent properties of control registers. Each register is specified by a table showing the field names and their layout. The annotations below each register denote bit numbers. Further details of control register layout are implementation dependent.

The ‘r’ field indicates bits that are reserved for future expansion of the architecture. When reading from a control register, software should not interpret the value of any reserved bits. When writing to a control register with reserved bits, software should write these bits using a value previously read from that register. If no appropriate previous value is available, then software should write reserved bits as 0.

The ‘e’ field indicates bits reserved for future expansion of the address space using a sign-extended convention. Expansion bits will read as a sign-extension of the highest implemented bit. Software should write a sign-extension of the highest implemented bit into expansion bits. This approach is necessary if software is to be executed on a future implementation with more implemented address space.

Power-on reset values for control registers are summarized in the following table.

Control Register	Field	Power-on Reset Value
SR	CD	0
	PR	0
	SZ	0
	FR	0
	FD	1
	WATCH	0
	STEP	0
	BL	1
	MD	1
	MMU	0
EXPEVT		0
RESVEC		0

Table 142: Power-on reset values for control registers

Control Register	Field	Power-on Reset Value
VBR		0
All other control registers and fields		UNDEFINED

Table 142: Power-on reset values for control registers

15.2.1 SR

The status register (SR) contains fields to control the behavior of instructions executed by the current thread of execution.

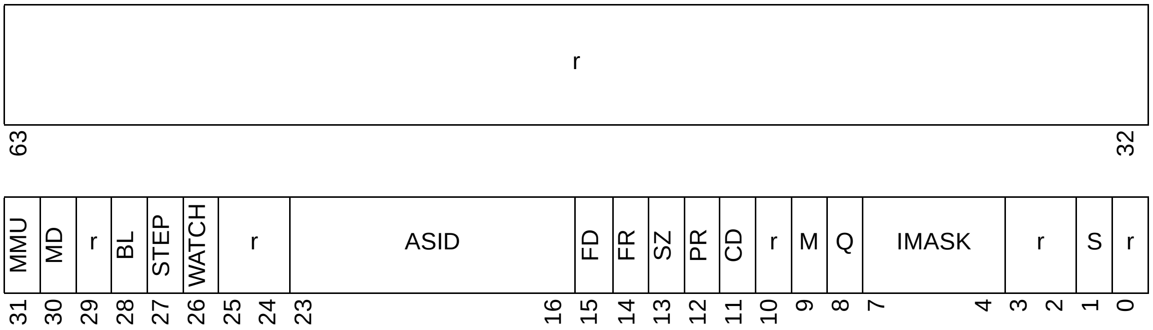


Figure 54: SR (upper 32 bits and lower 32 bits shown separately)

The fields of SR are summarized in the following table.

SR field	Accessibility for GETCON/PUTCON	Synopsis	Operation
S	RW	Saturation control	See *Chapter 11: SHcompact integer instructions on page 171*
Q	RW	State for divide step	
M	RW	State for divide step	
PR	RW	Floating-point precision	See *Chapter 13: SHcompact floating-point on page 189*
SZ	RW	Floating-point transfer size	
FR	RW	Floating-point register bank	
IMASK	RW	Interrupt request mask level	See *Chapter 16: Event handling on page 221*

Table 143: SR fields

SR field	Accessibility for GETCON/PUTCON	Synopsis	Operation
CD	RW	Clock tick counter disable flag	See *Section 15.2.10: CTC on page 218*
FD	If the implementation provides an FPU: RW If the implementation does not provide an FPU: RO	Floating-point disable flag	See *Section 8.2: Floating-point disable on page 135* and *Section 13.2: Floating-point disable on page 189*
ASID	RO (use RTE to write)	Address Space IDentifier	See *Chapter 17: Memory management on page 271*
WATCH	RO (use RTE to write)	Watch-point enable flag	See *Section 16.11.5: Debug exceptions on page 247*
STEP	RO (use RTE to write)	Single-step enable flag	See *Section 16.11.5: Debug exceptions on page 247*
BL	RW	Flag to block exception, trap or interrupt	See *Chapter 16: Event handling on page 221*
MD	RO (use RTE to write)	User (0) or privileged (1) mode	See *Chapter 2: Architectural state on page 13*
MMU	RO (use RTE to write)	MMU enable flag	See *Chapter 17: Memory management on page 271*

Table 143: SR fields

SR is explicitly read and written using GETCON and PUTCON. Additionally, SR is implicitly accessed as follows:

- SR is read during the execution of instructions.
- SR is written by some instructions (for example, SHcompact can write to S, Q and M).
- SR is written during launch sequences.
- SR is written by RTE (see *Chapter 16: Event handling on page 221*).

PUTCON can modify all fields of SR apart from ASID, WATCH, STEP, MD and MMU. Explicit writes to these particular fields have no effect.

The anticipated usage of SR fields is as follows. The ASID, WATCH, STEP, MD and MMU fields are managed asynchronously with respect to the current thread, perhaps by a kernel or by a debugger. The other SR fields are managed by the current thread when it is executing in privileged mode. They are typically modified by privileged threads using a GETCON to get the current SR value, some bit manipulation to change that value and a PUTCON to set the new SR value. This sequence of instructions is not necessarily atomic with respect to traps, exceptions and interrupts. Since the PUTCON does not affect the ASID, WATCH, STEP, MD and MMU fields, these fields are automatically preserved across such sequences. This property allows them to be managed independently without difficult or expensive software synchronization.

Unlike PUTCON, however, the RTE instruction can modify all defined fields within SR. Where it is necessary for a privileged thread to modify the ASID, WATCH, STEP, MD or MMU fields within SR, this can be achieved by placing appropriate values into SSR and SPC and using an RTE instruction.

SR.FD is 1 after a reset to indicate that floating-point is disabled. Software can distinguish whether the implementation provides a floating-point unit by writing 0 to SR.FD and reading its value back. If SR.FD reads as zero then a floating-point unit is provided, otherwise it is not.

Execution of a floating-point instruction with SR.FD set to 1 causes a floating-point disabled exception. If an implementation provides a floating-point unit, SR.FD must be cleared to allow floating-point instructions to be executed. If an implementation does not provide a floating-point unit, SR.FD is permanently set to 1.

Future implementations of the architecture could provide defined semantics for reserved bits of SR. It is therefore important that software preserves reserved bits of SR where possible. It is strongly recommended that SR is modified using instruction sequences equivalent to the following:

- Use GETCON to read SR into a temporary register.
- Modify necessary bits in the temporary register (leaving other bits unchanged).
- Use PUTCON to write SR from the temporary register.

For modifications to the ASID, WATCH, STEP, MD and MMU fields the above sequence should be adapted to write to SSR, so that RTE can then be used.

15.2.2 SSR

SSR is used to hold a saved copy of SR. The fields of SSR correspond to those in SR, except that every defined field is readable and writable.

SSR is used during launch and during return from launch sequences. During a launch sequence, hardware saves the previous SR into SSR. During a return from launch, hardware restores the current SSR into SR. For further information see *Section 16.6.2: Standard launch sequence on page 228* and *Section 16.7: Recovery on page 235*.

Future implementations of the architecture could provide defined semantics for reserved bits of the status register. It is therefore important that software preserves all reserved bits of SSR from the point of launch to the point of return from launch. This approach will improve compatibility with future implementations.

For an implementation that does not provide a floating-point unit, SR.FD is read-only and is permanently set to 1. However, the corresponding field in SSR (SSR.FD) will be readable and writable. If software sets SSR.FD to 0 on an implementation without an FPU and executes an RTE instruction, the behavior will be architecturally undefined.

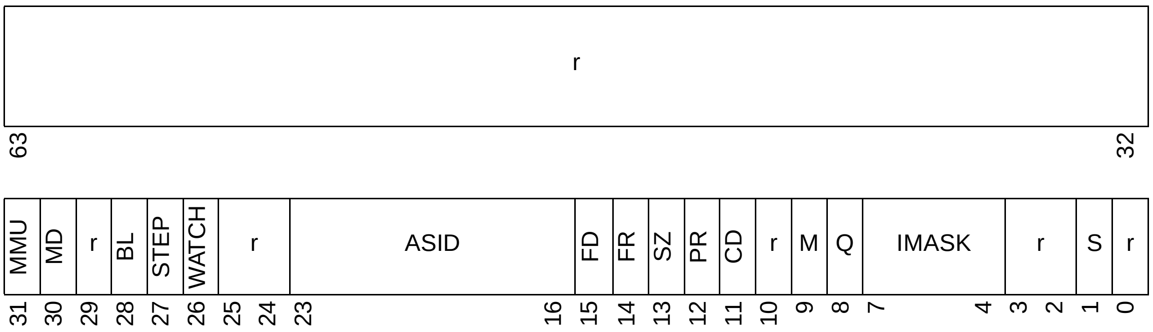


Figure 55: SSR (upper 32 bits and lower 32 bits shown separately)

15.2.3 PSSR

PSSR is used to hold a saved copy of SSR. The fields of PSSR correspond to those in SR and SSR. Every defined field is readable and writable.

PSSR is used during panic launch and during return from launch sequences. During a panic launch sequence, hardware saves the previous SSR into PSSR. During a return from launch, hardware restores the current PSSR into SSR. For further information see *Section 16.6.2: Standard launch sequence on page 228* and *Section 16.7: Recovery on page 235*.

Future implementations of the architecture could provide defined semantics for reserved bits of the status register. It is therefore important that software preserves all reserved bits of PSSR from the point of panic launch to the point of return from launch. This approach will improve compatibility with future implementations.

For an implementation that does not provide a floating-point unit, SR.FD is read-only and is permanently set to 1. However, the corresponding field in PSSR (PSSR.FD) will be readable and writable. If software sets PSSR.FD to 0 on an implementation without an FPU and executes an RTE instruction, the 0 value of PSSR.FD will be copied to SSR.FD as expected. However, if another RTE is executed with SSR.FD set to 0, then the behavior will become architecturally undefined as described in *Section 15.2.2: SSR on page 213*.

r	
63	32

MMU	MD	r	BL	STEP	WATCH	r	ASID	FD	FR	SZ	PR	CD	r	M	Q	IMASK	r	S	r
31	30	29	28	27	26	25 24	23 16	15	14	13	12	11	10	9	8	7 4	3 2	1	0

Table 144: PSSR (upper 32 bits and lower 32 bits shown separately)

15.2.4 INTEVT, EXPEVT, PEXPEVT, TRA

INTEVT is used to indicate the cause of the most recent interrupt. It is set during interrupt launch sequences.

EXPEVT is used to indicate the cause of the most recent reset, panic or exception. It is set during launch sequences.

PEXPEVT is used to hold the pre-panic value of EXPEVT. It is set during panic launch sequences.

TRA holds the operand value from a TRAPA instruction (see *Section 9.2: Event handling instructions on page 163*). When a TRAPA instruction is executed, the lower 32 bits of its operand value are loaded into TRA.

The launch sequence is described in *Section 16.6.2: Standard launch sequence on page 228*. These registers have the same representation shown in the following diagram. The CODE field is readable and writable.

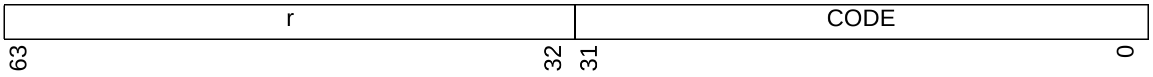


Figure 56: INTEVT, EXPEVT, PEXPEVT, TRA

15.2.5 SPC, PSPC

SPC is used to hold a saved copy of PC and ISA. It is used during launch and during return from launch sequences. For further information see *Section 16.6.2: Standard launch sequence on page 228* and *Section 16.7: Recovery on page 235*. Every defined field is readable and writable. The expansion field is reserved for future expansion of the address space.

PSPC is used to hold the pre-panic value of SPC. It is set during panic launch sequences. For further information see *Section 16.6.2: Standard launch sequence on page 228* and *Section 16.7: Recovery on page 235*. Every defined field is readable and writable. The expansion field is reserved for future expansion of the address space.

SPC and PSPC have the same representation shown in the following diagram.

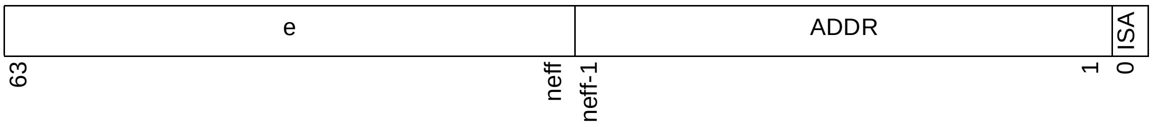


Figure 57: SPC, PSPC

If SPC indicates a misaligned SHmedia instruction (that is, if bits 0 and 1 are both set) and an RTE instruction is executed, then the behavior is architecturally undefined.

If bits 0 and 1 of PSPC are both set to 1 and an RTE instruction is executed, then these bits will be copied to SPC as expected. However, if another RTE instruction is executed with this value of SPC, the behavior will be architecturally undefined as described above.

15.2.6 RESVEC

RESVEC is used to determine the launch address at which execution is started after a reset, panic or debug event. A debug mechanism is provided that also allow these events to be re-vectored to a debugger. The lowest bit of RESVEC is used to indicate whether the MMU is automatically disabled during the launch. Further information is given in *Section 16.6.4: Handler addresses on page 231* and *Section 16.11.5: Debug exceptions on page 247*.

Every defined field is readable and writable. The expansion field is reserved for future expansion of the address space.

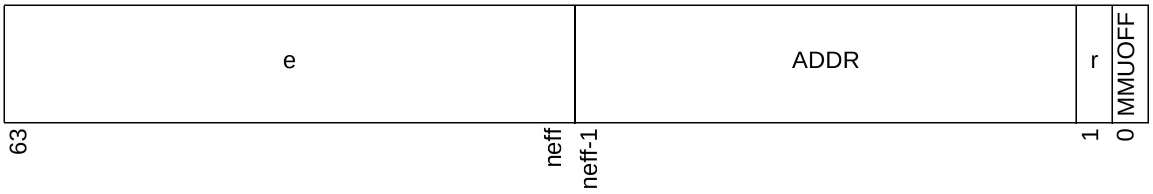


Figure 58: RESVEC

15.2.7 VBR

VBR is used to indicate the base address from which non-debug handler addresses are vectored. Further information is given in *Section 16.6.4: Handler addresses on page 231*. Every defined field is readable and writable. The expansion field is reserved for future expansion of the address space.

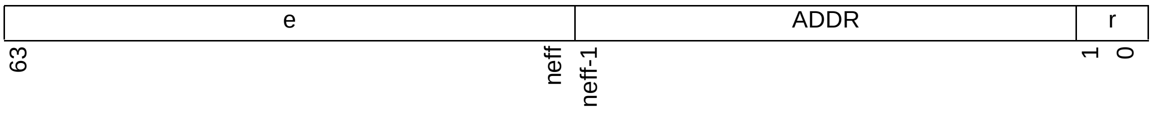


Figure 59: VBR

15.2.8 TEA

TEA is used to hold an effective address during some exception launches (see *Section 16.6.2: Standard launch sequence on page 228*). Every defined field is readable and writable. The expansion field is reserved for future expansion of the address space.

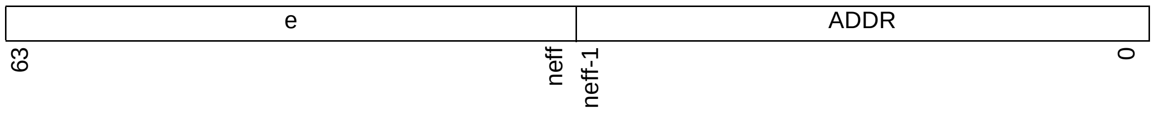


Figure 60: TEA

15.2.9 DCR, KCR0, KCR1

DCR is provided specifically for use by a debugger. The architecture makes no interpretation of the value held in this register. DCR is 64 bits wide, matching the width of general-purpose registers, and every bit is readable and writable. A debugger would typically use this register as a save location for a general-purpose register during debug event launch and restart. It is likely that debug software will not preserve the value of DCR across debug event handling. Thus software that is unrelated to debug should not access this register, since its value could be modified non-deterministically by debug software.

KCR0 and KCR1 are provided for the use of privileged mode software, such as an operating system kernel. The architecture makes no interpretation of the values held in these registers. They are 64 bits wide, matching the width of general-purpose registers, and every bit is readable and writable.

DCR, KCR0 and KCR1 have the same representation shown in the following diagram.

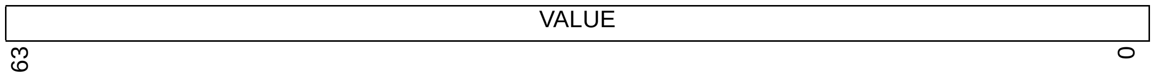


Figure 61: DCR, KCR0, KCR1

15.2.10 CTC

CTC is the clock tick counter. It is typically used for performance monitoring.

The number of implemented bits in CTC is implementation dependent. This value is denoted nctc and is in the range [32,64]. The frequency of the CPU clock is determined by an external clock reference and is system dependent. Also, the rate of execution of instructions is completely implementation dependent, and thus the values seen by reads from CTC will vary from implementation to implementation.

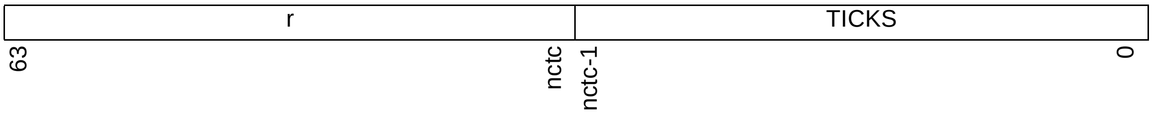


Figure 62: CTC

CTC has an undefined value after power-on reset. The counter should be initialized by software to give it a defined value.

When the CPU is not in sleep mode, CTC is decremented by 1 on every CPU clock cycle. The counter silently wraps around to its maximum value (which is implementation-dependent) when it decrements past zero. When the CPU is in sleep mode (see *Section 16.15: Power management on page 257*), the counter stops decrementing and its value does not change. No other mechanisms are provided for starting and stopping the counter.

The implemented bits of CTC are readable and writable in privileged mode. In user mode, a write to CTC never changes the value of CTC. In user mode, the behavior of reads from CTC is controlled by SR.CD. This allows privileged mode software to decide whether to allow a user thread to view the passing of time. If SR.CD is 0 then user-mode reads are enabled and return the value of CTC. If SR.CD is 1 then user-mode reads are disabled and always return 0.

The behavior of reads and writes to CTC are summarized in the following table.

SR.MD	SR.CD	Behavior for GETCON	Behavior for PUTCON
0 (user mode)	0 (enabled)	Returns current value	Write ignored
	1 (disabled)	Returns 0	Write ignored
1 (privileged mode)	0 (enabled)	Returns current value	Updates current value
	1 (disabled)		

Table 145: Behavior of CTC accesses

15.2.11 USR

USR is a user-accessible status register. It contains two fields, GPRS and FPRS, that can be used by software to track whether a register subset is in a dirty state or not.

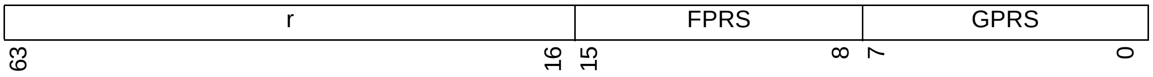


Figure 63: USR

The bits of GPRS and FPRS are used as follows for i in the range [0,7]:

- Bit i of USR.GPRS is the dirty bit for registers R_{8i} to R_{8i+7} inclusive.
- Bit i of USR.FPRS is the dirty bit for registers FR_{8i} to FR_{8i+7} inclusive.

Dirty bits can be read and written by software, either in user mode or in privileged mode.

When an instruction is executed that writes to a modifiable general-purpose register or floating-point register, then the dirty bit for the subset containing that register will be set to 1. However, note that a write to R_{63} is not required to set its dirty bit since the value of R_{63} is always 0.

The hardware can set dirty bits under other circumstances, but it never automatically clears the dirty bits. This is only achieved by explicit software action.

In the case of a GETCON from USR to a general-purpose register, the read of USR happens before the general-purpose register is written. The architecture does not require that the written general-purpose register is marked as dirty in the value read from USR. However, the architecture does require that the written general-purpose register is marked as dirty in a subsequent read from USR.

As an example, consider:

```
PUTCON R63, USR    ; clear USR
GETCON USR, R0     ; USR read happens before R0 write
GETCON USR, R1     ; USR read happens before R1 write
```

where USR denotes the user-accessible status register. The architecture does not require that the write to R_0 is visible in the value read from USR into R_0. However, the architecture does require that the write to R_0 is visible in the value read from USR into R_1.

The architecture leaves considerable flexibility in the implementation of the dirty bits. The only requirements are that all writes to modifiable general-purpose registers and floating-point registers cause the appropriate dirty bits to be set, and that a GETCON from USR is precise with respect to all such earlier updates.

An implementation is allowed to set dirty bits under other circumstances. Thus, a GETCON from USR can observe a set dirty bit that does not correspond to an earlier register modification. The important property is that writes to modifiable registers are never lost, though spurious writes can be reported.

Allowed implementation options include (but are not limited to):

- A write to R_{63} could set the appropriate dirty bit.
- GETCON from USR could mark the target register as dirty before reading USR.
- An implementation could update dirty bits imprecisely such that future writes to registers are reported speculatively. Possible situations that could lead to imprecise updates include (but are not limited to):
  - Event launches (see *Chapter 16: Event handling on page 221*): the dirty bits could be updated imprecisely for instructions that are partially executed, but do not complete (i.e. they are cancelled), due to the processor accepting an event and launching an event handler.
  - Branches: the dirty bits could be updated imprecisely for instructions that are partially executed, but do not complete (i.e. they are cancelled), due to speculative execution following a branch instruction.

  In general, note that the architecture maintains a precise architectural state (see *Section 16.4: Precision on page 227*). The USR control register is the exception to this rule.
- An implementation could implement larger subsets (that is, 4 subsets of 16 registers, 2 subsets of 32 registers or 1 subset of 64 registers) provided that all dirty bits corresponding to the larger subset are set appropriately.
- An implementation could choose not to implement any dynamic monitoring of register updates, and permanently set all of the dirty bits to 1.

If the FPU is disabled or is not present (i.e. SR.FD is set to 1), the USR.FPRS field is still implemented. GETCON can be used to read USR.FPRS and PUTCON can be used to update USR.FPRS to a new value. However, when SR.FD is 1 the implementation will not implicitly set any bits in USR.FPRS to indicate dirty registers. This is because all instructions that can modify floating-point registers will raise an exception when SR.FD is 1. Additionally, when SR.FD is 1 the implementation will not modify USR.FPRS imprecisely.

SuperH

Event handling 16

16.1 Overview

An event is a condition which requires the CPU to discontinue the normal execution of the current thread. Events can occur asynchronously or synchronously with respect to the execution of instructions.

When an event occurs, the processor stops executing the current program stream in order to respond to the event. Launch is the set of activities performed by the processor before execution of the next instruction. The processor saves some elements of the program state, and then arranges to execute instructions to handle the event. Full details of the launch sequence are described in *Section 16.6: Launch on page 228*.

A handler is the set of instructions that are executed in response to the event. The program counter of the first instruction of the handler (the handler address) is calculated during the launch sequence. This is achieved by adding some constant offset value onto the value of some base register. The selected offset and the selected base register depend on the event type. Further details of this calculation are described in *Section 16.6.4: Handler addresses on page 231*.

In many cases a handler completes execution by using the RTE instruction. This is described in *Section 16.7: Recovery on page 235*.

16.2 Asynchronous events

An asynchronous event is caused when some external condition is signaled to the CPU. Asynchronous events do not result directly from the execution of an instruction. The mechanisms used to deliver asynchronous events to the CPU are not specified by the CPU architecture. A typical mechanism is through signals delivered into the CPU core by wires.

There are two classes of asynchronous event: resets and interrupts.

16.2.1 Resets

A reset is an event that causes the current execution to be stopped, and to then be continued from some initial state. The mechanisms used to deliver reset events to the CPU are properties of the system architecture.

There are two kinds of reset distinguished by the CPU architecture and these are listed in *Table 146*.

Event Handle	Event Name
CPURESET	CPU Reset
POWERON	Power-on Reset

Table 146: Reset handles

A power-on reset results from the first application of power to the CPU device, and causes the CPU to initialize itself into a specific power-on reset state. A power-on reset is special in that the CPU has no previous state.

A CPU reset is used to reset the CPU device while it is powered on. Previous state is available after CPU reset allowing analysis of the pre-reset state of the system. However, not all of the previous state is preserved. Restart of the previous program state is typically neither possible nor attempted.

The system architecture provides two further kinds of reset:

- MANUAL reset behaves in the same way as a POWERON reset, except that some parts of the memory system are preserved. This provides a power-on reset without loss of values held in volatile memory.
- DEBUG reset behaves in the same way as a POWERON reset, except that some items of debug state are preserved. This allows a complete system to be debugged immediately after it comes up from its power-on state.

The effect of these two resets, as far as the CPU core architecture is concerned, is the same as a power-on reset. The additional effects of these resets are specified by the system architecture.

16.2.2 Interrupts

An interrupt is an event resulting from some external stimulus, perhaps delivered through an interrupt pin from some external device. Interrupts are typically used to bring to the attention of the CPU the fact that some interesting condition has arisen on some other part of the system.

Event Handle	Event Name
DEBUGINT	Debug Interrupt
EXTINT	External Interrupt
NMI	Non-maskable Interrupt

Table 147: Interrupt handles

After an interrupt the previous program state is usually restartable. Restart can be achieved without disturbing the interrupted program (apart from the latency of the handling). However, there is one special form of interrupt, called a non-maskable interrupt, which can result in some architectural state being lost. It is not generally possible to restart the previous program state after a non-maskable interrupt.

16.2.3 Assertion, deassertion and acceptance

Specific terminology is used to describe the delivery of asynchronous events. The status of an asynchronous event is delivered from its *source* to the CPU using a *signal*. If the event requires the attention of the CPU, then this signal is *asserted*, otherwise the signal is *deasserted*.

When a signal has an asserted value, and the CPU is able to launch a handler for that event, the event is said to be *accepted* by the CPU. The CPU checks the acceptance condition of each signal between the execution of consecutive instructions. If a signal is asserted but cannot be accepted by the CPU at a particular instant, it is still possible for that asserted signal to be accepted at some future instant provided that the signal is asserted at that future time. Acceptance of an event causes a handler to be launched for that event.

The acceptance point of an asynchronous event is non-deterministic. The event is handled between some pair of instructions chosen by the implementation in a timing-dependent manner. Implementations should endeavor to minimize the latency of asynchronous events (in particular, interrupts) to improve real-time behavior.

The mechanisms that cause signals to be asserted and deasserted are implementation-specific. There are 2 common arrangements:

- *Level-triggered event*: the assertion/deassertion status of the signal is derived directly from the source of the event. Acceptance of this event by the CPU does not implicitly deassert the value of the signal. Typically, software action is needed to deassert the signal value (for example, by memory accesses).
- *Edge-triggered event*: the signal becomes asserted each time that an edge is detected on the source of the event. The details of how this edge is detected are implementation-specific. Acceptance of this event by the CPU causes the signal to become automatically deasserted. No software action is needed to deassert the signal value.

16.3 Synchronous events

A synchronous event is caused by the execution of an instruction. A synchronous event is handled before or after the execution of the instruction that caused that event. There are two classes of synchronous event: exceptions and panics. Both of these events are caused by abnormal program behavior, but they differ in the severity of the problem.

16.3.1 Exceptions

Exception can usually be recovered from to allow program execution to be continued.

Event handle	Event name
BREAK	Software break
DEBUGIA	Instruction Address Debug Exception
DEBUGIV	Instruction Value Debug Exception
ITLBMISS	Instruction TLB Miss Error
RADDERR	Data Address Error Read
READPROT	Data TLB Protection Violation Read

Table 148: Exception handles

Event handle	Event name
DEBUGOA	Operand Address Debug Exception
DEBUGSS	Single Step Debug Exception
EXECPROT	Instruction Protection Violation
FPUDIS	FPU Disabled Exception
FPUEXC	FPU Exception
IADDERR	Instruction Address Error
ILLSLOT	Illegal Slot Exception
RESINST	Reserved Instruction Exception
RTLBMISS	Data TLB Miss Read
SLOTFPUDIS	Delay Slot FPU Disabled Exception
TRAP	Unconditional trap
WADDERR	Data Address Error Write
WRITEPROT	Data TLB Protection Violation Write
WTLBMISS	Data TLB Miss Write

Table 148: Exception handles

The exception mechanism supports the translation capabilities of the memory management unit. When an access is made to a page that has no entry in the translation lookaside buffer (TLB), a TLB miss exception is taken. Further details are given in *Chapter 17: Memory management on page 271*.

16.3.2 Panics

A panic results from a serious software failure. Software systems are typically constructed to avoid panics. The panic handling mechanism is provided as a debug aid to allow panic situations to be debugged. The panic mechanism can also be used to debug critical code sections such as exception and interrupt launch.

Event handle	Event name
PANIC	Panic

Table 149: Panic handles

16.3.3 Pre-execution and post-execution

There are two points when a synchronous event can be delivered relative to the execution of the instruction that causes that event.

If the event is delivered before the instruction updates architectural state, then it is a pre-execution synchronous event. The program counter, at the point where the event is taken, refers to the instruction that caused the event.

If the event is delivered after the instruction updates architectural state, then it is a post-execution synchronous event. The program counter, at the point where the event is taken, refers to the next instruction that would normally execute after the instruction that caused the event. Thus, this 'next' instruction is the instruction that would have executed next if the post-execution event had not happened. There is no architectural guarantee that this 'next' instruction will ever be executed since that depends on the actions of the launched exception handler. Even if that handler restarts execution at that 'next' instruction, another event could occur before that 'next' instruction causing yet another handler launch.

There is only one post-execution synchronous event in the architecture. This is used to support single-stepping of instructions for debugging. In this case, post-execution is the natural choice since it allows instructions to be executed one at a time with minimal software overheads.

All other synchronous events in the architecture are pre-execution. In most cases this is the natural choice. Often, software handles the condition that caused the event and restarts the originating instruction. Where software requires restart at the instruction after the originating instruction, it is necessary for software to calculate the next instruction pointer. Typically, this is a very simple calculation such as incrementing the delivered program counter by the instruction size.

The distinction between pre-execution and post-execution is not useful for asynchronous events. Asynchronous events occur between instructions. Whether an asynchronous event is said to occur after the previous instruction or before the next instruction is arbitrary.

SHcompact provides instructions with architecturally-visible delayed branching. There are special rules for the meaning of pre-execution and post-execution for these branches. These rules are described in *Section 16.6.3*.

16.4 Precision

The launch of interrupt, exception and panic handlers is precise with respect to the instruction stream prior to the launch. The launch happens exactly between two instructions from that stream. All previous instructions from that stream have completed execution and updated the architecturally-visible state. No subsequent instruction from that stream have updated the architecturally-visible state.

These properties are upheld from the point of view of instructions executing on the CPU. An implementation can make substantial optimization over this model provided that the architecturally-visible properties are upheld.

For power-on reset, precision has no meaning since there is no previous instruction stream. The launch of a CPU reset handler is not guaranteed to be precise. The previous state of the machine observed by a CPU reset handler is not guaranteed to be completely accurate.

Note that the architecture allows an implementation to update the USR control register imprecisely, and that this is an exception to the architectural rules on precision (see *Section 15.2.11: USR on page 219*).

16.5 Debug and non-debug events

The architecture distinguishes between debug and non-debug events. This distinction is orthogonal to the other categorizations. Debug events can be vectored separately to non-debug events, to allow a debugger to be implemented independently of other target software.

The asynchronous debug events are CPURESET and DEBUGINT. The synchronous debug events are BREAK, DEBUGIA, DEBUGIV, DEBUGOA, DEBUGSS and PANIC. All other events are non-debug.

Debug events can be launched using either the debug vector (DBRVEC) or launched using the reset vector (RESVEC). The selection between DBRVEC and RESVEC is controlled through a flag called DBRMODE. Debug events also have the capability to disable the MMU, and hence the caches, during the launch sequence. These properties allow debug software to be highly decoupled from other system software, and gives a powerful non-intrusive debug architecture.

Further information on DBRVEC and DBRMODE is given in *Section 16.6.4: Handler addresses on page 231*.

16.6 Launch

This section describes the activities associated with a launch sequence. These activities are mostly described in a generic way, though there are important differences between reset, interrupt, exception and panic launches. The particular conditions which cause different events to arise are described separately in *Section 16.9*, *Section 16.10*, *Section 16.11* and *Section 16.12*.

Many of the control registers are used when responding to an event, and these are described in *Chapter 15: Control registers on page 207*. The launch sequence sets control registers to save the previous thread state, to characterize the event and to launch a handler. Control registers are read to determine where the instructions of the handler are to be fetched from.

The state that is established for each kind of launch is summarized in *Section 16.14*.

16.6.1 Power-on reset launch sequence

The actions for a power-on reset are:

1 Set all initialized state to its power-on reset values (as defined in *Section 16.9: Resets on page 236*). The power-on reset value of the PC is 0.

2 Execution of instructions starts in SHmedia mode.

On some implementations it is possible to change the value of the PC between step *1* and step *2*, so that execution of instructions starts at an address which differs from the power-on reset value of the PC. Whether this mechanism is supported and how it is achieved are properties of the system architecture and not defined here.

16.6.2 Standard launch sequence

When an event occurs, the processor initiates a launch sequence to allow execution of a handler. This sequence preserves some of the original thread context, and then establishes a suitable thread context for running the handler. The standard launch sequence is used for all events apart from power-on reset, which is described separately in *Section 16.6.1*.

For CPU resets, interrupts, exceptions and panics, the processor does the following:

1 Determine the context of the original thread for the point at which the event is taken. The preciseness of this context is consistent with the properties stated in *Section 16.4*. For synchronous events the choice between pre-execution and

post-execution is determined by the event type as described in *Section 16.3.3*. Further information on special cases is given in *Section 16.6.3*.

2 If the event is a panic, the SSR, SPC and EXPEVT of the original thread are saved into PSSR, PSPC and PEXPEVT respectively. This additional saving of state allows the panic handler to reconstruct the conditions that led to the panic.

3 The PC, ISA and SR of the original thread are saved:

3.1 The PC and ISA are saved into SPC. The lowest bit of SPC holds ISA (a “1” indicates SHmedia while a “0” indicates SHcompact) and the remainder of SPC holds the PC. The SR of the original thread is saved into SSR.

Note that the saves to SSR, SPC, PSSR and PSPC are mirrored by the restores performed by the RTE instruction (see *Section 16.7: Recovery on page 235*).

4 Establish an appropriate context for the handler using the following assignments:

4.1 Ensure the handler will execute in privileged mode by setting SR.MD to 1.

4.2 Ensure the handler will execute in SHmedia by setting ISA to 1.

4.3 Ensure the handler will execute with blocking by setting SR.BL to 1. Nested launches for further exceptions and blockable interrupts do not occur until the handler has had an opportunity to save some necessary state.

4.4 If this launch is for a debug event, SR.STEP and SR.WATCH are cleared. This ensures that single-step and watch-point exceptions are not taken during debug handling. Otherwise, this launch is for a non-debug event, and SR.STEP and SR.WATCH are preserved. Non-debug handlers automatically inherit the single-step and watch-point behavior of the excepting or interrupting thread. This allows non-debug handlers to be debugged using the single-step, watch-point and panic mechanism.

4.5 If this launch is for a CPU reset, set SR.CD, SR.FD, SR.FR, SR.SZ and SR.PR to their CPU reset values.

4.6 If the event is a maskable interrupt, set SR.IMASK to the level of this interrupt. This ensures that only higher priority interrupts will be accepted while handling this interrupt. This mechanism supports efficient nested interrupts. Note that no maskable interrupts will be accepted while interrupts are blocked by SR.BL.

4.7 Determine the appropriate value of SR.MMU (described in *Section 16.6.5*).

5 Provide information to characterize the event through the following assignments:

5.1 If the event is an external interrupt or NMI, write the appropriate code to INTEVT. If the event is a debug interrupt, then INTEVT is not changed.

5.2 If the event is not an interrupt, write the appropriate event code to EXPEVT.

5.3 If the event is a trap, write the value of the trap operand to TRA.

5.4 If the event is an exception that delivers an address, write that address to TEA.

5.5 If the event is a floating-point execution exception, set FPSCR to indicate the cause of the exception.

If the event is a panic, note that EXPEVT is updated but that INTEVT, TRA, TEA and FPSCR are preserved.

6 Determine the handler address (described in *Section 16.6.4*) and set PC to it.

7 Execution of instructions continues in SHmedia at the new PC.

16.6.3 Launch point

During launch, the processor stores an address in the SPC. In most cases, SPC indicates the address and ISA to which the handler should return after handling the event. The SPC contents depend on the type of the event and the instruction on which the event is taken. *Table 150* describes the behavior.

Special cases arise for the SHcompact delayed branch mechanism. When a delayed branch is executed, any resulting flow of control to the branch destination is delayed by 1 instruction. The instruction that sequentially follows the delayed branch in the program text is called the delay slot. The delay slot is executed regardless of whether the branch is taken. SHcompact contains conventional and delayed branches.

An SHcompact delayed branch instruction and its delay slot are executed indivisibly with respect to events. Events are either taken before the delayed branch or after the delay slot. An event is never taken between the delayed branch and the delay slot. Pre-execution events that occur on either instruction are taken before the delayed branch. Post-execution events that occur on either instruction are taken after the delay slot, and the delivered SPC depends on whether the branch was taken or not.

SHmedia does not have delayed branches. The special cases arise in SHcompact only.

This can occur in mode:	Exception occurs	SPC contents for a pre-execution event	SPC contents for a post-execution event
SHmedia/SHcompact	Not in delay-slot and not on a branch	Address of instruction	Address of following instruction
SHmedia/SHcompact	On non-delayed, untaken branch	Address of branch instruction	Address of following instruction
SHmedia/SHcompact	On non-delayed, taken branch	Address of branch instruction	Address of target instruction
SHcompact only	On delayed, untaken branch	Address of branch instruction	Address of instruction following delay slot
SHcompact only	On delayed, taken branch	Address of branch instruction	Address of target instruction
SHcompact only	In delay-slot of untaken branch	Address of branch instruction	Address of instruction following delay slot
SHcompact only	In delay-slot of taken branch	Address of branch instruction	Address of target instruction

Table 150: SPC contents

16.6.4 Handler addresses

A handler address is calculated by adding a base register value with an offset.The architecture provides three base registers: VBR, RESVEC and DBRVEC. The choice of base register and offset value depends on the event type, and is specified in *Table 151*.

Event Type		Base Register		Offset
		(DBRMODE=0)	(DBRMODE=1)	
Power-on reset		0x0		0x0
Non-debug exception	Not a TLB miss	VBR		0x100
	TLB miss	VBR		0x400
Non-debug interrupt		VBR		0x600
CPU reset or panic		RESVEC	DBRVEC	0x0

Table 151: Handler start addresses

Event Type	Base Register		Offset
	(DBRMODE=0)	(DBRMODE=1)	
Debug exception	RESVEC	DBRVEC	0x100
Debug interrupt	RESVEC	DBRVEC	0x200

Table 151: Handler start addresses

The control register containing the base register is read as a 64-bit value. The two least-significant bits of this value are then forced to 0 to ensure 4-byte alignment. Note that the lowest bit of RESVEC and DBRVEC is used to control MMU disabling (see *Section 16.6.5*) and is masked off for the handler address calculation.

No checking is performed on the addition of the base register value with the offset. The base registers and the offsets are architected so that this calculation can never result in a misaligned instruction pointer. If the addition of the base register with the offset results in an address outside of the implemented effective address space, the behavior is architecturally undefined. This situation must be avoided by software.

VBR

The vector base register (VBR) is used for all non-debug exceptions and non-debug interrupts. Different offsets are used for the following 3 cases:

- Non-debug exceptions not pertaining to translation look-up misses.
- Non-debug exceptions pertaining to translation look-up misses.
- Non-debug interrupts.

This separation reduces the handler latency for translation look-up misses and interrupts. It also allows unrelated handling code to be decoupled from each other. Translation look-up is described in *Chapter 17: Memory management on page 271*.

VBR is a control register and can be read and written using GETCON and PUTCON (see *Section 15.2.7: VBR on page 216*).

RESVEC and DBRVEC

Debug events are vectored using either the reset vector base register (RESVEC, see *Section 15.2.6: RESVEC on page 216*) or the debug base register vector (DBRVEC). The selection between RESVEC and DBRVEC is specified through the DBR mode (DBRMODE). The debug events are listed in *Section 16.5* and include debug exceptions, debug interrupts, CPURESET and PANIC.

RESVEC is a control register and can be read and written using GETCON and PUTCON. RESVEC is given the value 0x0 after a power-on reset. RESVEC is not used to vector power-on reset. The power-on reset vector is always 0x0.

DBRVEC and DBRMODE are memory-mapped registers. The CPU architecture states that these two registers exist and specifies how the values of these registers affect debug event launch. Other details of these registers, including their addresses and layout, are defined separately by the system architecture.

DBRMODE controls whether debug events are vectored through RESVEC (when DBRMODE is 0) or DBRVEC (when DBRMODE is 1). The power-on values of DBRVEC and DBRMODE are implementation-specific.

The vector offsets used by RESVEC and DBRVEC are symmetrical. CPU reset and panic use one offset, debug exceptions use a second offset, and debug interrupts use a third offset (as specified in *Table 151*). These offsets allow separation of different handlers.

RESVEC is provided for the use of privileged mode software running on the processor (for example, an operating system). DBRVEC and DBRMODE are provided for the debugger. By programming DBRVEC and DBRMODE, a debugger can attach debug handlers to the processor without interaction with other software running on the processor.

16.6.5 Effect of launch on MMU and caches

The status bit SR.MMU controls whether the MMU and caches are disabled or enabled. After a power-on reset, SR.MMU is 0. Launches for all non-debug exceptions and non-debug interrupts leave SR.MMU unchanged. Launches for CPU reset and panic events always clear SR.MMU.

Launches for debug exceptions and debug interrupts have a programmable capability to disable the MMU, and hence the caches, during the launch sequence. This is a debug-specific feature that is not provided for non-debug events.

This feature is controlled by the lowest bit of the base register used to vector the debug event. When DBRMODE is 0, the lowest bit of RESVEC (called RESVEC.MMUOFF) is used. When DBRMODE is 1, the lowest bit of DBRVEC (called DBRVEC.MMUOFF) is used. There are 2 cases:

- If MMUOFF is zero, then the value of SR.MMU is not changed by a launch for a debug event.
- If MMUOFF is one, then the value of SR.MMU is automatically cleared by a launch for a debug event. This has the effect of disabling the MMU and also

disabling the caches. This behavior is described in *Section 17.8.1: Cache behavior when the MMU is disabled on page 289*. The previous value of SR.MMU is saved in SSR.MMU. This allows the previous state of the MMU to be recovered.

Note that the value of MMUOFF is ignored for CPU reset and panic. These debug events always clear SR.MMU regardless of MMUOFF.

16.6.6 Event codes

Every event is characterized by an event code. The event codes used by the architecture are specified in *Section 16.13: Event ordering and event summary tables on page 251*.

Event codes are held in INTEVT for interrupts, and in EXPEVT for other events. Additionally, PEXPEVT is used to save EXPEVT during a panic launch. These 3 registers each provide 32 bits of state. They are initialized automatically by hardware during launch sequence, and can also be read and written by software.

The event codes that are assigned by the architecture are listed in *Section 16.13: Event ordering and event summary tables on page 251*. These assigned codes have the format shown in *Table 152*. This means that all assigned event codes are in the range [0x000, 0x1000) and are exact multiples of 0x20. Note that only a subset of the 128 event codes in this range are currently assigned.

0	7-bit value	0
31 12	11 5	4 0

Table 152: Assigned event code format

Event codes in the range [0, 0x80000000) can be used by future versions of the architecture to distinguish more kinds of event. Future architectures will assign event codes such that the least significant 5 bits are 0, and will therefore maintain the property that all event codes are exact multiples of 0x20. Event codes in the [0, 0x80000000) range that are currently unassigned are reserved. They must not be used for software purposes.

Event codes in the range [0x80000000, 0x100000000) are not assigned by the current architecture and will not be assigned by future versions of the architecture. They are available for use by software, and can be used to distinguish different kinds of software event.

16.7 Recovery

The RTE instruction (see *Section 9.2: Event handling instructions on page 163*) allows a handler to recover a previous program context. This is often used as the final instruction of a handler. RTE performs the following actions:

1. PC and ISA are restored from SPC. The lowest bit of SPC determines the ISA mode of the next instruction to be executed. The remaining bits of SPC determine the program counter of the next instruction to be executed.
2. SR is restored from SSR.
3. SPC is restored from PSPC, and SSR is restored from PSSR. These restores allow recovery from panic events.
4. Execution of instructions continues from PC in the mode indicated by ISA.

RTE results in architecturally-undefined behavior if the values of SPC and SSR are inappropriate:

- Execution of RTE when SPC.ISA is 1 and lowest bit of SPC.ADDR is 1: this setting corresponds to a misaligned SHmedia instruction and is not supported.
- Execution of RTE when SSR.FD is 0 on an implementation without a floating-point unit: this setting corresponds to an attempt to enable the FPU when it is not supported.

16.8 Instruction synchronization

Event launch and recovery (RTE) synchronize the instruction stream. The implementation implicitly performs the same instruction synchronization as defined for SYNCI (see *Section 6.5.2: Instruction synchronization on page 99*).

The next instruction after a launch or after an RTE is fetched correctly according to the architectural state in force at that time. For example, a launch or RTE can cause the following major changes to the architectural state:

- The MMU can be enabled or disabled.
- The privilege level MD can change.
- The ASID can change (RTE only).
- The ISA mode can change.

The implementations of launch and RTE ensures that the next instruction is fetched correctly with respect to such changes. Note that MMU, MD and ASID changes can cause substantial changes to address translation, and that this is correctly handled by the implementation.

16.9 Resets

The CPU architecture distinguishes power-on reset and CPU reset.

Power on reset (POWERON)

Cause	External to the CPU core. This occurs when power is first applied.
Actions	CPU is initialized to its power-on reset state (see following table) and begins executing the code located at address 0

State	Value
ISA (implicit state)	1 (execution in SHmedia)
PC	0 (this is the power-on reset value of RESVEC)
General purpose registers, R	R0 to R62 are UNDEFINED, R63 reads as zero
Target registers, TR	UNDEFINED
Floating-point registers, FR	UNDEFINED
FPSCR	UNDEFINED
MEM	Power-on memory state is not specified by CPU architecture
SR.WATCH	0 (watch-points disabled)
SR.BL	1 (blocked)
SR.MD	1 (privileged execution)
SR.MMU	0 (MMU and cache disabled)
SR.FD	1 (FPU disabled)

Table 153: Power-on reset state

State	Value
SR.CD, SR.FR, SR.SZ, SR.PR	0 (default values)
SR.STEP	0 (single-step disabled)
EXPEVT	0 (event code to indicate a power-on reset)
RESVEC	0
VBR	0
All other control register fields	UNDEFINED
All configuration registers	UNDEFINED
Data and instruction cache state	UNDEFINED

Table 153: Power-on reset state

CPU reset (CPURESET)

Cause	External to the CPU core. CPU reset preserves much of the state of the CPU, and causes an event launch through RESVEC or DBRVEC according to DBRMODE. It is reasonable to consider CPU reset as being a special non-restartable interrupt.
Actions	CPU is initialized to its CPU reset state (see following table) and begins executing the code located at the address specified in RESVEC.

State	Value
ISA (implicit state)	1 (execution in SHmedia)
PC	If DBRMODE is 0: PC is RESVEC If DBRMODE is 1: PC is DBRVEC
General purpose registers, R	UNCHANGED
Target registers, TR	UNCHANGED
Floating-point registers, FR	UNCHANGED

Table 154: CPU reset state

State	Value
FPSCR	UNCHANGED
MEM	UNCHANGED
SR.WATCH	0 (watch-points disabled)
SR.BL	1 (blocked)
SR.MD	1 (privileged execution)
SR.MMU	0 (MMU and cache disabled)
SR.FD	1 (FPU disabled)
SR.CD, SR.FR, SR.SZ, SR.PR	0 (default values)
SR.STEP	0 (single-step disabled)
EXPEVT	0x20 (event code to indicate a CPU reset)
SPC	Saved PC (holds the PC prior to the CPU reset)
SSR	Saved SR (holds the SR prior to the CPU reset)
All other control register fields	UNCHANGED
All configuration registers	UNCHANGED
Data and instruction cache state	UNCHANGED

Table 154: CPU reset state

CPU reset is not classed as an exception and therefore does not cause a panic when CPU reset occurs while SR.BL is set. This means that SPC, SSR and EXPEVT are never saved to PSPC, PSSR and PEXPEVT (respectively) on a CPU reset launch. The prior state of SPC, SSR and EXPEVT is lost upon CPU reset.

A CPU reset does not cancel pending interrupts. Using the terminology of *Section 16.2.3*, if an interrupt signal is asserted while conditions do not allow that interrupt to be accepted and a CPU reset occurs, then the CPU reset does not automatically cause the interrupt signal to be deasserted. If the interrupt source continues to assert the signal and conditions subsequently allow that interrupt to be accepted, then the interrupt event will be taken.

Specifically, this means that pending EXTINT and DEBUGINT interrupts which happen to be masked or blocked (as appropriate) are not cancelled if a CPU reset occurs. If the interrupt source continues to assert the signal, the interrupt event will be taken after the CPU reset should conditions allow. Note that NMI can be neither masked nor blocked so NMI interrupts are never held pending. Assertion of an NMI interrupt will be detected before or after any concurrent CPU reset, and will be handled before or after that CPU reset, depending on the relative timing of these two asynchronous events.

16.10 Interrupts

This section describes the properties of the CPU architecture for handling interrupts.

Two mechanisms are provided for controlling when an asserted interrupt is accepted by the CPU and a handler launched for it:

- Blocking is achieved by setting SR.BL. When SR.BL is set, blockable interrupts are not accepted by the CPU.
- Masking is achieved through the interrupt mask field (SR.IMASK). This specifies the priority level of the CPU. A maskable interrupt is not accepted by the CPU if the interrupt's priority level is less than or equal to the CPU's priority level.

Three kinds of interrupt are distinguished by the CPU.

Event handle	Event name	Priority	Maskable?	Blockable?
NMI	Non-maskable interrupt	17	No	No
DEBUGINT	Debug interrupt	16	No	Yes
EXTINT	External interrupt	0 to 15	Yes	Yes

Table 155: Interrupts

The mechanisms used to generate interrupts are implementation specific.

16.10.1 Non-maskable interrupt

There is a single source of non-maskable interrupts (NMI). When this interrupt is asserted, it causes the launch of a handler at the next available gap between executing instructions. An NMI is accepted regardless of the state of the CPU, even if other interrupts are blocked or masked. This is potentially a destructive operation since the launch can lose some of the pre-interrupt context (for example, SPC and SSR). In general, an NMI handler should not attempt to return to the pre-interrupt context.

NMI has a priority level of 17, and therefore takes priority over any other pending interrupts. This priority level is greater than any CPU priority level, and NMI is accepted regardless of the value of SR.IMASK.

The event code associated with NMI is specified by the CPU architecture. The value of SR.IMASK is not changed when a handler is launched for NMI.

Non-maskable interrupt (NMI)

Cause	External to the CPU core.
Actions	Standard launch sequence is followed. INTEVT is set with the event code of the interrupt. This event is asynchronous and can be neither blocked nor masked.

16.10.2 Debug interrupt

There is a single source of debug interrupts (DEBUGINT). Debug interrupts are blocked when SR.BL is set, but cannot be masked.

DEBUGINT has a priority level of 16. NMI has a higher priority than DEBUGINT, but DEBUGINT has a higher priority than all other interrupts. This priority level is also greater than any CPU priority level, and a debug interrupt is accepted regardless of the value of SR.IMASK.

There is no event code associated with a debug interrupt, and the value of INTEVT is not changed during the launch sequence for a debug interrupt. The base register and offset used for calculating the handler address for debug interrupts are not used by other events (see *Section 16.6.4: Handler addresses on page 231*). This allows debug interrupts to be distinguished from other all events without relying on an event code.

The value of SR.IMASK is not changed when a handler is launched for DEBUGINT.

Debug interrupt (DEBUGINT)

Cause	External to the CPU core.
Actions	Standard launch sequence is followed. This is a debug event and is vectored through RESVEC or DBRVEC according to DBRMODE.This event is asynchronous. It can be blocked but cannot be masked.

16.10.3 External interrupts

External interrupts are maskable and are assigned a priority level in the range [0, 15]. The lowest priority interrupt has the lowest value (0) and the highest priority interrupt has the highest value (15). An asserted external interrupt is accepted when all of the following conditions are true:

- It is the highest priority pending interrupt.
- SR.BL is zero.
- The priority level of the interrupt is greater than the CPU's priority level.

An interrupt with priority level 0 is never accepted since it cannot have a priority level greater than that of the CPU. A priority level 0 interrupt can cause the CPU to exit sleep mode (see *Section 16.15.2: Exiting sleep mode on page 258*) even though the interrupt will never be accepted and will never cause a handler to be launched.

The event code associated with an external interrupt is not specified by the CPU architecture. When an external interrupt is accepted by the CPU, the source of the interrupt signal (which is external to the CPU core) provides an event code to identify the interrupt. This code is loaded into INTEVT during the launch sequence.

The processor priority level (SR.IMASK) is set to the priority level of the interrupt. This supports nested interrupt handling in an efficient manner.

External interrupt (EXTINT)

Cause	External to the CPU core.
Actions	Standard launch sequence is followed. INTEVT is set with the event code of the interrupt. This event is asynchronous and can be both blocked and masked.

16.11 Exceptions

Exceptions are categorized into instruction address exceptions, instruction opcode exceptions, data address exceptions, floating-point exceptions and debug exceptions.

If an exception occurs while SR.BL is 1, it is treated as a panic (see *Section 16.12*). All exceptions are pre-execution unless otherwise stated.

16.11.1 Instruction address exceptions

Instruction address exceptions are related to the memory management features described in *Chapter 17: Memory management on page 271*.

Instruction TLB miss (ITLBMISS)

Cause	Fetch from an address which does not have a translation
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception. TEA is set with the faulty instruction address

Instruction protection violation (EXECPROT)

Cause	Instruction fetch from a prohibited page. In privileged mode this occurs if the fetch address is in a non-executable page. In user mode this occurs if the fetch address is in a non-executable page or a page that is not accessible from user mode.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception. TEA is set with the faulty instruction address.

Instruction address error (IADDERR)

SHmedia	Preparation of a target register with an instruction address that is misaligned or not in the implemented effective address space.
SHcompact	Branch to an instruction address that is misaligned or not in the implemented effective address space.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception. TEA is set with the faulty instruction address. The number of implemented bits in TEA matches the number of bits in effective addresses. If the faulty address is not in the implemented effective address space, then upper bits of the faulty address will be lost.

16.11.2 Instruction opcode exceptions

These exceptions are detected by decoding the instruction opcode.

Illegal slot exception (ILLSLOT)

SHmedia	Never occurs
SHcompact	The instruction is in a delay slot, and the instruction is one of the following: a reserved instruction, an instruction that modifies PC (any branch or RTS), a PC-relative move instruction, MOVA or TRAPA.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception.

Unconditional trap (TRAP)

SHmedia	Execution of the TRAPA instruction. The SHmedia TRAPA instruction is described in *Section 9.2: Event handling instructions on page 163*.
SHcompact	Execution of a TRAPA instruction that is not in a delay slot. The SHcompact TRAPA instruction is described in *Section 14.1: System instructions on page 205*.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception. The operand of the TRAPA instruction is loaded into the TRA register (see *Section 15.2.4: INTEVT, EXPEVT, PEXPEVT, TRA on page 215*).

Reserved instruction exception (RESINST)

SHmedia	Execution of a reserved instruction or of a privileged-mode instruction while in user mode. The full set of reserved SHmedia instructions are specified in *Volume 2, Appendix A: SHmedia instruction encoding*. Note that the SHmedia instruction with encoding 0x6FF4FFF0 is guaranteed to be reserved on all implementations.
SHcompact	Execution of a reserved instruction that is not in a delay slot. The full set of reserved SHcompact instructions are specified in *Volume 3, Appendix A: SHcompact instruction encoding*. Note that the SHcompact instruction with encoding 0xFFFD is guaranteed to be reserved on all implementations.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception.

16.11.3 Data address exceptions

Data address exceptions are related to the memory management features described in *Chapter 17: Memory management on page 271*.

Read address error (RADDERR)

Cause	Load using a misaligned address, or using an address that is not in the implemented effective address space.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception. TEA is set with the faulty data address. The number of implemented bits in TEA matches the number of bits in effective addresses. If the faulty address is not in the implemented effective address space, then upper bits of the faulty address will be lost.

Write address error (WADDERR)

Cause	Store using a misaligned address, or using an address that is not in the implemented effective address space.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception. TEA is set with the faulty data address. The number of implemented bits in TEA matches the number of bits in effective addresses. If the faulty address is not in the implemented effective address space, then upper bits of the faulty address will be lost.

Read TLB miss (RTLBMISS)

Cause	Load using an address which does not have a translation.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception. TEA is set with the faulty data address.

Write TLB miss (WTLBMISS)

Cause	Store using an address which does not have a translation.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception. TEA is set with the faulty data address.

Read protection violation (READPROT)

Cause	Read access from a prohibited page. In privileged mode this occurs if the read address is in a non-readable page. In user mode this occurs if the read address is in a non-readable page or a page that is not accessible from user mode.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception. TEA is set with the faulty data address.

Write protection violation (WRITEPROT)

Cause	Write access to a prohibited page. In privileged mode this occurs if the written address is in a non-writable page. In user mode this occurs if the written address is in a non-writable page or a page that is not accessible from user mode.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception. TEA is set with the faulty data address.

16.11.4 FPU exceptions

Three exceptions are used by the floating-point architecture.

FPU disabled exception (FPUDIS)

SHmedia	Execution of a floating-point instruction while the floating-point unit is disabled. The set of SHmedia floating-point instructions is defined in *Volume 2, Appendix A: SHmedia instruction encoding*. All floating-point instructions have a zero value in bits [0, 3] of their encoding. If these bits are non-zero, the instruction is never considered a floating-point instruction and its execution will cause a RESINST exception.
SHcompact	Execution of a floating-point instruction while the floating-point unit is disabled. This exception is not taken if the instruction is in a delay slot. The set of SHcompact floating-point instructions are specified in *Volume 3, Appendix A: SHcompact instruction encoding*.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception.

Delay-slot FPU disabled exception (SLOTFPUDIS)

SHmedia	Never occurs.
SHcompact	Execution of a floating-point instruction in a delay slot while the floating-point unit is disabled.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception.

FPU execution exception (FPUEXC)

Cause	The FPU is enabled, and execution of an FPU instruction raises a floating-point exception.
Actions	Standard launch sequence is followed. EXPEVT is set with the event code of the exception. FPSCR is set to indicate the reason for the floating-point exception.

16.11.5 Debug exceptions

This section describes the debug exceptions. There are 5 debug exceptions that support 3 independent debug mechanisms: watch-point, break and single-step.

Watch-points

The debug architecture uses watch-points to detect 3 different exceptions for instruction fetch and data access:

- Debug instruction address (IA) exceptions are detected by comparing the fetched instruction address against watch-point address ranges.
- Debug instruction value (IV) exceptions are detected by comparing the instruction encoding against watch-point instruction patterns.
- Debug operand address (OA) exceptions are detected by comparing accessed data addresses against watch-point address ranges.

Watch-point detection is disabled when SR.WATCH is 0, and enabled when SR.WATCH is 1. The mechanisms used to generate watch-point exceptions are properties of the system architecture.

Break

The BRK instruction (see *Section 9.2: Event handling instructions on page 163*) unconditionally raises a debug exception and is useful for implementing soft break-points. The BRK instruction is typically reserved for use by the debugger.

Single-step

The single-step mechanism is used to trigger a debug exception after the execution of each instruction. Unlike all other exceptions, single-step is post-execution.

Single-step exceptions are raised when SR.STEP is 1. SR.STEP can only be changed explicitly by the RTE instruction; a PUTCON to SR does not change SR.STEP. Reset, panic and debug launches automatically clear SR.STEP to prevent inappropriate single-stepping of those handlers. Other launches preserve SR.STEP so that those handlers can be single-stepped.

There is only one way in which the single-step condition can arise. This is when an RTE instruction is executed with SSR.STEP set. The RTE instruction causes SR to be restored from SSR, while PC is restored from SPC. The instruction referred to by PC is executed, the PC is updated to the next instruction, and then a single-step exception is raised.

If SR.BL is 0 this results in a debug handler being launched, while if SR.BL is 1 then a panic handler is launched instead. In both of these cases it is possible to restart the program after the single-stepped instruction.

The values of SR.STEP and SR.BL that are used to make these decisions are sampled after the RTE is executed but before the single-stepped instruction is executed. If the single-stepped instruction changes SR.STEP and/or SR.BL (that is, it is an RTE or a PUTCON) then these changes do not effect the launch of the single-step event following that instruction. In other words, the values of SSR.STEP and SSR.BL seen by an RTE instruction (and restored into SR.STEP and SR.BL) determine the single-step post-execution behavior of the next instruction executed after the RTE.

All other exceptions take precedence over single-step. This follows naturally since single-step is the only post-execution exception. Consider the behavior when an instruction is executed when SR.STEP is set. If that instruction raises a pre-execution exception then that will take precedence over the single-step. However, the value of SR.STEP is saved into SSR, so that if the instruction is restarted and it completes without a pre-execution exception, then the single-step will be correctly taken.

It is possible to single-step any instruction including the RTE instruction itself. The architected behavior follows from the above description. There are some particularly interesting single-step cases and these are described in *Section 16.16: Single-step behavior on page 261*.

When executing in SHcompact, a delayed branch and its delay slot are executed indivisibly. A single-step exception is not taken between these 2 instructions. When single-stepping through a delayed branch and its delay slot, a single-step exception will be taken with the SPC referring to the delayed branch and the next will be taken with the SPC referring to the instruction that executes after the delay slot (which instruction depends on whether the branch is taken or not).

Debug instruction address exception (DEBUGIA)

Cause	Execution of an instruction when SR.WATCH is 1 and the instruction address triggers a debug IA watch-point.
Actions	Standard launch sequence. This is a debug event and is vectored through RESVEC or DBRVEC according to DBRMODE. EXPEVT is set with the event code of the exception. TEA is not set. Software must instead deduce the triggering instruction address from the value of SPC.

Debug instruction value exception (DEBUGIV)

Cause	Execution of an instruction when SR.WATCH is 1 and the instruction encoding triggers a debug IV watch-point.
Actions	Standard launch sequence. This is a debug event and is vectored through RESVEC or DBRVEC according to DBRMODE. EXPEVT is set with the event code of the exception. TEA is set with the address of the matched instruction.

Debug operand address exception (DEBUGOA)

Cause	Execution of an instruction when SR.WATCH is 1 and the operand address triggers a debug OA watch-point.
Actions	Standard launch sequence. This is a debug event and is vectored through RESVEC or DBRVEC according to DBRMODE. EXPEVT is set with the event code of the exception. TEA is set with the matched operand address.

Break (BREAK)

SHmedia	Execution of the BRK instruction.
SHcompact	Execution of the BRK instruction (even if in a delay slot).
Actions	Standard launch sequence is followed. This is a debug event and is vectored through RESVEC or DBRVEC according to DBRMODE. EXPEVT is set with the event code of the exception. A BRK will cause a pre-execution debug exception. For more information on the BRK instruction see *Section 9.2: Event handling instructions on page 163*. The BRK instruction is typically reserved for use by the debugger.

Debug single step exception (DEBUGSS)

Cause	Execution of an instruction when SR.STEP is 1. This is a post-execution exception, and is raised after the single-stepped instruction completes.
Actions	Standard launch sequence is followed. This is a debug event and is vectored through RESVEC or DBRVEC according to DBRMODE. EXPEVT is set with the event code of the exception.

16.12 Panics

If an exception occurs while SR.BL is 1, a panic launch is initiated rather than an exception launch. This rule applies to all exceptions, including debug exceptions.

Normally, software is designed so that exceptions do not occur while SR.BL is 1. If this situation arises, it is usually indicative of a serious system fault. The panic feature is provided as a debugging aid to allow the conditions that led to the panic to be reconstructed and analyzed. The single-stepping mechanism uses panic to allow single-stepping into code sequences with SR.BL is 1. This allows, for example, single-step debugging of exception and interrupt handlers.

The architecture provides panic control registers to save the state modified by the panic launch. This approach ensures that the pre-panic state is available. It is possible to return from a panic without loss of architectural state (apart from the state specifically provided for panic handling).

Panic (PANIC)

Cause	An exception occurs while SR.BL is 1.
Actions	A panic launch is pre-execution if the exception that caused the panic was pre-execution. A panic launch is post-execution if the exception that caused the panic was post-execution. The only post-execution panic occurs when a single-step exception is taken when SR.BL is 1. Standard launch sequence is followed. The MMU is disabled. The PSPC, PSSR and PEXPEVT control registers are used to preserve the values of SPC, SSR and EXPEVT prior to the panic. SPC and SSR are used to preserve the values of PC and SR prior to the panic. EXPEVT is set with the event code of the exception that caused the panic. This event code is always distinct from the codes used for POWERON and CPURESET.

16.13 Event ordering and event summary tables

It is possible for multiple events to occur at the same time. This section defines the arbitration between these multiple events to choose which event should result in a launch. This is achieved by specifying an ordering between those events. When multiple events occur, the event with the lowest order number is taken.

Event ordering is specified in *Table 156* and *Table 157*. These tables also give the abbreviated names (handles), handler addresses and event codes for every event. The acceptance point of asynchronous events is non-deterministic with respect to instruction execution, and with respect to the detection of synchronous events.

16.13.1 Ordering of asynchronous events

The ordering of asynchronous events is shown in the following table. The event with the lowest order number has precedence. There are multiple external interrupts and these are ordered by priority level (see *Section 16.10.3*). For external interrupts, the interrupt event code is provided by the source of the interrupt signal.

Event handle	Event name	Order	Vector[a]	Offset	EXPEVT or INTEVT
POWERON	Power-on Reset	1	0x0	0x0	0x000 (EXPEVT)
CPURESET	CPU Reset	2	VEC	0x0	0x020 (EXPEVT)
NMI	Non-maskable Interrupt	3	VBR	0x600	0x1C0 (INTEVT)
DEBUGINT	Debug Interrupt	4	VEC	0x200	Not changed
EXTINT	External Interrupt	5	VBR	0x600	Various (INTEVT)

Table 156: Ordering of asynchronous events

a. If DBRMODE is 0, VEC is RESVEC, otherwise VEC is DBRVEC.

16.13.2 Ordering of synchronous events

The ordering of synchronous events is shown in the following table. The event with the lowest order number has precedence. Where two or more events share an order number those events can never occur simultaneously.

Event handle	Event name	Order	Vector[a]	Offset	EXPEVT	Pre or Post
PANIC	Panic	1	VEC	0x0	Various[b]	Pre/Post[c]
DEBUGIA	Instruction Address Debug	2	VEC	0x100	0x900	Pre
ITLBMISS	Instruction TLB Miss Error	3	VBR	0x400	0xA40	Pre
EXECPROT	Instruction Protection Violation	4	VBR	0x100	0xAA0	Pre
DEBUGIV	Instruction Value Debug	5	VEC	0x100	0x920	Pre
FPUDIS	FPU Disabled	6	VBR	0x100	0x800	Pre
SLOTFPUDIS	Delay Slot FPU Disabled	6	VBR	0x100	0x820	Pre
BREAK	Software break	7	VEC	0x100	0x940	Pre
TRAP	Unconditional trap	7	VBR	0x100	0x160	Pre
RESINST	Reserved Instruction	7	VBR	0x100	0x180	Pre
ILLSLOT	Illegal Slot Exception	7	VBR	0x100	0x1A0	Pre
IADDERR	Instruction Address Error[d]	8	VBR	0x100	0xAE0	Pre
DEBUGOA	Operand Address Debug	9	VEC	0x100	0x960	Pre
RADDERR	Data Address Error Read	10	VBR	0x100	0x0E0	Pre
WADDERR	Data Address Error Write	10	VBR	0x100	0x100	Pre
RTLBMISS	Data TLB Miss Read	11	VBR	0x400	0x040	Pre
WTLBMISS	Data TLB Miss Write	11	VBR	0x400	0x060	Pre
READPROT	Data Protection Violation Read	12	VBR	0x100	0x0A0	Pre
WRITEPROT	Data Protection Violation Write	13	VBR	0x100	0x0C0	Pre
FPUEXC	FPU Exception	14	VBR	0x100	0x120	Pre
DEBUGSS	Single Step Debug	15	VEC	0x100	0x980	Post

Table 157: Ordering of synchronous events

a. If DBRMODE is 0, VEC is RESVEC, otherwise VEC is DBRVEC.

b. The event code for a panic depends upon the code of the event that causes the panic.

c. Panic is pre- or post-execution depending upon the disposition of the event causing the panic.

d. IADDERR occurs during execution of SHmedia prepare-target and SHcompact branch instructions.

16.13.3 SHcompact event ordering

There are 2 cases where the SHcompact priority order differs from the normal order.

Instructions with 2 memory accesses

Some SHcompact instructions make 2 accesses to memory; exceptions are detected separately for these 2 accesses. The affected instructions are shown in the table.

SHcompact instruction	First access	Second access
MAC.L @Rm+, @Rn+	Read from @Rn+	Read from @Rm+
MAC.W @Rm+, @Rn+		
AND.B #imm, @(R0, GBR)	Read from @(R0, GBR)	Write to @(R0, GBR)
OR.B #imm, @(R0, GBR)		
XOR.B #imm, @(R0, GBR)		
TAS.B @Rn	Read from @Rn	Write to @Rn

Table 158: SHcompact instructions with multiple accesses

Exception checking is achieved in the following order for read/read cases:

1 RADDERR, then RTLBMISS, then READPROT for first read access

2 Memory is read for first read access

3 RADDERR, then RTLBMISS, then READPROT for second read access

4 Memory is read for second read access

Exception checking is achieved in the following order for read/write cases:

1 RADDERR, then RTLBMISS, then READPROT for read access

2 Memory is read

3 WRITEPROT for write access

4 Memory is written

In both cases, note that memory is read for the first access before exception conditions are checked for the second access. This means that exception checking is not completely precise on these instructions. Care is required if these instructions are used on memory locations that have side-effects on reads (for example, some devices have this property).

Delayed branch and delay slots

In general, SHcompact delayed branches and delay slots are executed indivisibly with respect to event launch. Exception checking is achieved in the following order:

1 The delayed branch is checked for pre-execution exceptions (that is, all exceptions apart from single-step). If an exception arises, the exception handler is launched based on the state prior to the execution of the delayed branch and the delay slot.

2 The delay slot is checked for all pre-execution exceptions (that is, all exceptions apart from single-step). If an exception arises, the exception handler is launched based on the state prior to the execution of the delayed branch and the delay slot.

3 The delayed branch is executed and updates architectural state.

4 The delay slot is executed and updates architectural state.

5 The single-step exception is checked after the execution of both instructions. If an exception arises, the exception handler is launched based on the state after the execution of both the delayed branch or the delay slot.

There is a special case which deviates from the behavior described above. If the delayed branch instruction writes to PR and does not raise a pre-execution exception, but the delay slot instruction does raise a pre-execution exception, then PR will be updated before the exception handler is launched.

In this case the execution of the delayed branch and the delay slot is not completely indivisible, and the state observed by the exception handler is not completely precise. The state delivered to the exception handler, apart from PR, is consistent with the state prior to the execution of either the delayed branch or the delay slot.

This special case affects the following instructions:

- BSR label
- BSRF Rn
- JSR @Rn

Thus, where an exception is raised on the delay slot of one of these instructions, PR is updated with the procedure link value (the PC of the instruction following the delay slot) before the exception handler is launched. Note that these instructions do not read the value of PR. If the exception handler can fix the cause of the exception, then it can safely restart execution at the delayed branch instruction without affecting the correct behavior of the original thread.

16.14 Launch assignments

This section summarizes the changes made to the architectural state during launch.

Every launch makes the following assignments:

- ISA is set to 1 to indicate execution in SHmedia.
- SR.MD is set to 1 to indicate privileged execution.
- SR.BL is set to 1 to indicate blocking.
- PC is set to the handler address (as indicated in *Table 156* and *Table 157*).

Additionally, a launch can optionally update SPC, SSR, EXPEVT, INTEVT, TEA, TRA, SR.MMU, SR.STEP, SR.WATCH, SR.IMASK, FPSCR, PSSR, PSPC and PEXPEVT. These assignments are summarized in *Table 159* and *Table 160*. The entry in each cell of the table is one of the following:

UN: this state has an undefined value after this launch.

0: this state is always initialized to zero by this launch.

✔: this state is initialized as described in *Section 16.6.2: Standard launch sequence on page 228*.

(no entry): this state is not initialized by this launch (that is, it is preserved).

16.14.1 Asynchronous launch

Handle	SPC SSR	EXPEVT	INTEVT	TEA	TRA	SR.STEP SR.WATCH	SR.IMASK	SR.MMU	FPSCR	PSSR PSPC PEXPEVT
POWERON	UN	✔	UN	UN	UN	0	UN	0	UN	UN
CPURESET	✔	✔				0		0		
NMI	✔		✔							
DEBUGINT	✔					0		✔		
EXTINT	✔		✔				✔			

Table 159: Launch assignments for asynchronous events

Power-on and CPU reset set SR.CD, SR.FD, SR.FR, SR.SZ and SR.PR to their reset values. Additionally, power-on reset sets RESVEC and VBR to their reset values.

16.14.2 Synchronous launch

Handle	SPC SSR	EXPEVT	INTEVT	TEA	TRA	SR.STEP SR.WATCH	SR.IMASK	SR.MMU	FPSCR	PSSR PSPC PEXPEVT
PANIC	✔	✔				0		0		✔
DEBUGIA	✔	✔				0		✔		
ITLBMISS	✔	✔		✔						
EXECPROT	✔	✔		✔						
DEBUGIV	✔	✔		✔		0		✔		
BREAK	✔	✔				0		✔		
TRAP	✔	✔			✔					
RESINST	✔	✔								
ILLSLOT	✔	✔								
FPUDIS	✔	✔								
SLOTFPUDIS	✔	✔								
DEBUGOA	✔	✔		✔		0		✔		

Table 160: Launch assignments for synchronous events

Handle	SPC SSR	EXPEVT	INTEVT	TEA	TRA	SR.STEP SR.WATCH	SR.IMASK	SR.MMU	FPSCR	PSSR PSPC PEXPEVT
IADDERR	✔	✔		✔						
RADDERR	✔	✔		✔						
WADDERR	✔	✔		✔						
RTLBMISS	✔	✔		✔						
WTLBMISS	✔	✔		✔						
READPROT	✔	✔		✔						
WRITEPROT	✔	✔		✔						
FPUEXC	✔	✔							✔	
DEBUGSS	✔	✔				0		✔		

Table 160: Launch assignments for synchronous events

16.15 Power management

The CPU supports two modes of operation. In normal mode, the CPU executes instructions as usual. In sleep mode, execution of instructions is suspended. On implementations that support power-down of the CPU, the power consumed by the CPU will be significantly reduced while in sleep mode.

The transition from normal mode to sleep mode is achieved by the execution of the SLEEP instruction. The transition from sleep mode to normal mode is triggered by the assertion of an interrupt or a CPU reset. The state of the CPU is preserved across sleep periods. In some cases the exit from sleep mode is associated with an event handler launch, and this launch causes the standard set of launch assignments to the architectural state.

On implementations that provide an FPU and support independent power-down of the FPU, there will also be a power-saving when the FPU is disabled. It is therefore recommended that the FPU is disabled, by setting SR.FD to 1, where it is known that floating-point instructions are not required.

Other power management properties are implementation-specific.

16.15.1 Entering sleep mode

Sleep mode is entered by executing the SLEEP instruction. This is a privileged instruction. Execution of this instruction in user mode will cause a reserved instruction exception.

Instruction	Summary
SLEEP	enter sleep mode

Table 161: Sleep instruction

Upon execution of the SLEEP instruction, the CPU is placed into sleep mode and stops executing further instructions until sleep mode is exited. On entry to sleep mode all earlier instructions including the SLEEP instruction have completed execution, and no subsequent instructions have started execution.

SLEEP does not automatically synchronize instruction fetch and data accesses. *Section 16.15.4* describes the necessary actions to achieve this synchronization.

The SLEEP instruction does not modify the blocking status (SR.BL) nor the interrupt mask level of the CPU (SR.IMASK). These two fields continue to control the blocking and masking of interrupt launch in the usual way.

16.15.2 Exiting sleep mode

At the point of sleep, the SLEEP instruction is considered to have completed execution, though any post-execution exception conditions on that instruction will not yet have been checked. The PC at the point of sleep is the address of the instruction immediately following the SLEEP instruction. This property ensures that an interrupt launch after the sleep will deliver an SPC to allow restart at the instruction that immediately follows the SLEEP instruction.

Sleep mode is exited when any asynchronous event source is asserted. This is regardless of whether conditions allow that event to be accepted and cause a handler launch. For example, assertion of an external interrupt at priority level 0 causes the CPU to exit sleep mode, even though the interrupt will never be accepted and will never cause a handler to be launched (see *Section 16.10.3: External interrupts on page 241*).

When the CPU leaves sleep mode, the SLEEP instruction is checked for post-execution exceptions before any handler launch for the asynchronous wake-up event. This is because asynchronous conditions are checked between instructions: that is, after the post-execution checks of the previous instruction but before the pre-execution checks of the following instruction. After post-execution exceptions have been checked, the CPU checks for asynchronous conditions.

The asynchronous events that can cause an exit from sleep are CPURESET, NMI, DEBUGINT and EXTINT. These cause an appropriate handler launch as soon as conditions allow the event to be accepted. CPURESET and NMI cannot be blocked, and immediately cause a handler launch. The handler launch for DEBUGINT and EXTINT can be delayed due to blocking or masking.

The different cases are:

1 CPURESET: a reset handler is launched.

2 NMI: an NMI handler is launched.

3 DEBUGINT: a debug interrupt handler is launched if interrupts are not blocked. If interrupts are blocked, then the handler is not launched and execution continues with the next instruction after the SLEEP instruction.

4 EXTINT: an interrupt handler is launched if the interrupt is not blocked and not masked. If the interrupt is blocked or is masked, then the handler is not launched and execution continues with the next instruction after the SLEEP instruction.

An asserted DEBUGINT or EXTINT causes the CPU to exit sleep mode regardless of whether that interrupt causes a launch. The interrupt will be accepted when it is both asserted and the CPU is ready to accept that interrupt. Software will typically arrange for SR.BL to be cleared or for SR.IMASK to be reduced, as appropriate, so that the wake-up interrupt can be accepted.

If it is desired for only a subset of the possible wake-up events to wake the CPU, then this is achieved by controlling the generation of those events at their source. Alternatively, the SLEEP instruction can be iterated to repeatedly put the CPU into sleep mode until the desired condition is reached.

Handler launch uses the standard launch mechanism. No specific indication is given to the handler that the CPU was previously in sleep mode. However, the interrupt event code (INTEVT) can be used to distinguish different interrupt causes.

Architectural state, including memory, is preserved across sleep periods. Exit from sleep can involve an event handler launch and this will cause assignments to state.

16.15.3 Sleep and wake-up timing

Wake-up signals are asynchronous events and are therefore asserted asynchronously with respect to instruction execution. The CPU checks the assertion of asynchronous events between the execution of consecutive instructions. This means that when a wake-up signal transitions from deasserted to asserted, this will be detected by the CPU either before or after a SLEEP instruction.

If the CPU detects an asserted wake-up signal after a SLEEP instruction, then the CPU will wake-up. If the event can be accepted, a handler is launched.

If the CPU detects an asserted wake-up signal before a SLEEP instruction and that event can be accepted, then a handler is launched. If the event is not accepted then no handler is launched. If the SLEEP instruction is executed and there is already an asserted asynchronous event, then the CPU will wake-up immediately after the execution of the SLEEP. In this case, the SLEEP instruction has the same architectural effect as NOP.

It is common that a sequence containing a SLEEP instruction is executed with interrupts blocked. This ensures that a launch for a blockable interrupt is delayed until after the SLEEP instruction has executed. Only once interrupts are unblocked can that launch be made.

16.15.4 Sleep and synchronization

The SLEEP instruction provides only limited synchronization of instruction fetches and data accesses.

At the point where the CPU enters sleep mode, the CPU arranges that there are no outstanding transactions in the memory system outside of the CPU core. However, it is possible that there could be earlier memory accesses which have not yet initiated such a memory transaction. To ensure that all previous memory accesses have completed out to external memory, software must use a SYNCO instruction prior to the SLEEP.

The SLEEP instruction ensures that all previous instructions have completed execution. However, it does not guarantee any synchronization of subsequent instruction fetch:

1. In cases where sleep exit causes a handler launch, instruction fetch is implicitly synchronized by the launch mechanism.
2. In cases where sleep exit does not cause a handler launch and instead restarts the instruction after the SLEEP, the hardware does not guarantee instruction fetch synchronization and this must be achieved by software. SYNCI must be

executed immediately after the SLEEP instruction to cause the necessary synchronization.

For correct synchronization, SLEEP must be used in the following code sequence:

```
SYNCO   ; synchronize data accesses
SLEEP   ; enter sleep mode
SYNCI   ; synchronize instruction fetch
```

16.16 Single-step behavior

An abstract model of instruction execution illustrates the single-step behavior. An instruction can be executed using the following actions:

1 If an interrupt or reset signal can be accepted, then update state to launch a handler. In this case, the current instruction becomes the first instruction of the launched handler.

2 If there is a pre-execution panic or exception condition on the current instruction, then update state so as to launch a handler. In this case, the current instruction becomes the first instruction of the launched handler.

3 Save SR.STEP in some temporary non-architected state (for example, saved_step).

4 Save SR.BL in some temporary non-architected state (for example, saved_bl).

5 Execute the current instruction (that is, the one at PC). The architectural state is updated and the PC is advanced to the next instruction. The calculation of the next instruction does not take into account any subsequent event launch.

6 If the current instruction is a SLEEP instruction, then enter sleep mode. Sleep mode is exited when an asynchronous event is asserted, as described in *Section 16.15: Power management on page 257*.

7 If saved_step is set, a post-execution single-step exception launch is required:

   If saved_bl is clear, then update state so as to launch a debug handler.

   If saved_bl is set, then update state so as to launch a panic handler.

   The current instruction becomes the first instruction of the launched handler.

8 Goto step 1 to execute another instruction.

A key property of this model is that asynchronous events are accepted *between* instructions. They are checked after the post-execution checks of the previous instruction but before the pre-execution checks of the following instruction.

The above description is illustrative and not intended to constrain implementations beyond the stated architectural requirements. Other models of execution are possible, though they must be indistinguishable from the above model from an architectural perspective.

16.16.1 Single-step across handler launch and RTE

There are two very interesting cases: single-step across a handler launch and single-step across an RTE. These cases are particularly important because of the ability to single-step in critical regions. The resulting behavior is described in the following example.

Let a program contain instructions I_a, I_b and I_c. Previous instructions have arranged for single-stepping through the program. Let instruction I_b cause an exception handler to be launched before its execution. Let the invoked handler have instructions I_l, I_m and I_n (where n is RTE)[1]. The handler fixes up the exception, and then performs an RTE back to I_b such that I_b can execute successfully. An alternative example would be the case of an interrupt being accepted before the execution of I_b; this would exhibit the same single-stepping behavior to that described here.

Let the debugger install single-step and panic handlers with instructions I_x, I_y and I_z (where I_z is RTE), and simply restart at the delivered SPC with single-step still enabled.

The architecture specifies that single-stepping is propagated into the exception handler. The launch of a debug handler or panic handler, however, always clears SR.STEP so these handlers will not be single-stepped.

The sequence of instructions that will be executed is:

1 Previous instructions are executed such that SR.STEP is now set

2 Execute I_a

Launch DEBUGSS with SPC=I_b: execute I_x, execute I_y, execute I_z (RTE to I_b)

1. In these examples, the number of instructions in handlers is minimized for ease of presentation. Typically, more instructions will be used to fix-up an exception.

3 Pre-execution event causes handler launch with SPC=I_b: execute I_l

Launch PANIC with SPC=I_m: execute I_x, execute I_y, execute I_z (RTE to I_m)

4 Execute I_m

Launch PANIC with SPC=I_n: execute I_x, execute I_y, execute I_z (RTE to I_n)

5 Execute I_n (RTE to I_b)

Launch PANIC with SPC=I_b: execute I_x, execute I_y, execute I_z (RTE to I_b)

6 Execute I_b

Launch DEBUGSS with SPC=I_c: execute I_x, execute I_y, execute I_z (RTE to I_c)

7 Subsequent instructions are executed

The numbered lines in this sequence show instructions being executed in the original program or in the exception handler. The unnumbered lines show instructions being executed in the debugger. Note that the debugger sees exactly one single-step exception between every executed instruction.

The order of SPC values seen by the debugger is: I_a I_b I_m I_n I_b I_c. The non-debugger instructions actually executed are: I_a I_l I_m I_n I_b I_c. Thus, I_b is repeated and I_l is not observed.

This behavior is explained as follows. The SPC delivered to the debugger refers to the next instruction. Next instruction means the next instruction that would be executed in the normal flow of instructions; that is, assuming that no event launch will immediately follow the current instruction. This is logical because single-step is a post-execution event of the current instruction, and therefore happens before the CPU has an opportunity to take a following interrupt or any exception on the next instruction.

When the debugger sees a particular SPC and then restarts that instruction, there is no guarantee that the SPC instruction will complete: there could be an interrupt or exception that causes a handler launch and a different instruction to be executed.

In this example instruction I_b raises a pre-execution exception, and instruction I_l is executed in its place. Debug software should be aware of this anomalous behavior.

16.16.2 Single-step and interrupts

Single-step exceptions are always raised on the instruction following an RTE.

If block and mask conditions allow, it is possible for an interrupt to be accepted after the RTE but before the following instruction executes and thus before a single-step can be delivered for that following instruction. Using the rules already described, an interrupt handler will be launched. For an NMI or EXTINT interrupt, this interrupt handler will inherit the single-step bit of the interrupted thread and in this case will therefore execute with SR.STEP set. This means that the single step condition will be present after the first instruction of the interrupt handler is executed. Since the launch of the interrupt handler will also have caused SR.BL to be set, this single-step event will be delivered as a panic. The launch of a debug handler (including DEBUGINT) or panic handler, however, always clears SR.STEP so these handlers will not be single-stepped. The overall effect is that single stepping propagates into NMI and EXTINT interrupt handlers.

In general, the architecture makes no statements about the timing of interrupts from their assertion in the outside world to their acceptance by the CPU. The CPU architecture describes the acceptance point of interrupts as non-deterministic giving an implementation considerable leeway. Clearly interrupts should be propagated in a timely manner to achieve good real-time behavior, but issues of timing are beyond the CPU architecture.

However, the architecture requires that an implementation must allow interrupts to be accepted between the RTE and a following single-stepped instruction (where blocking and masking allow). This ensures that interrupts can be delivered in the context of a single-stepped program.

If an implementation always delayed interrupts such that this never occurred, then single-step debugging could become functionally intrusive. The worst case is when the single-step handler itself runs completely with interrupts blocked; in this scenario EXTINT and DEBUGINT interrupts would never be accepted. This arrangement is avoided by declaring that such an implementation is not a valid realization of the architecture.

16.16.3 Single-step and sleep

Single-stepping through a sleep sequence is another interesting case.

Let a program contain instructions I_a, I_b and I_c. Previous instructions have arranged for single-stepping through the program. Let instruction I_b be the SLEEP instruction, and let an EXTINT interrupt subsequently become asserted and return the CPU to normal mode. Let this interrupt be neither blocked nor masked, so that it is possible to launch a handler for it. Let the invoked interrupt handler have instructions I_l, I_m and I_n (where n is RTE). The handler performs an RTE back to I_c.

Let the debugger install single-step and panic handlers with instructions I_x, I_y and I_z (where I_z is RTE), and simply restart at the delivered SPC with single-step still enabled.

The sequence of instructions that will be executed is:

1 Previous instructions are executed such that SR.STEP is now set

2 Execute I_a

Launch DEBUGSS with SPC=I_b: execute I_x, execute I_y, execute I_z (RTE to I_b)

3 Execute I_b and enter sleep mode

After some cycles, an interrupt causes exit from sleep mode

Launch DEBUGSS with SPC=I_c: execute I_x, execute I_y, execute I_z (RTE to I_c)

4 Interrupt condition causes handler launch with SPC=I_c; execute I_l

Launch PANIC with SPC=I_m: execute I_x, execute I_y, execute I_z (RTE to I_m)

5 Execute I_m

Launch PANIC with SPC=I_n: execute I_x, execute I_y, execute I_z (RTE to I_n)

6 Execute I_n (RTE to I_b)

Launch PANIC with SPC=I_b: execute I_x, execute I_y, execute I_z (RTE to I_c)

7 Subsequent instructions are executed

The numbered lines in this sequence show instructions being executed in the original program or in the exception handler. The unnumbered lines show instructions being executed in the debugger. Note that the debugger sees exactly one single-step exception between every executed instruction.

When instruction I_b executes it causes the CPU to be placed into SLEEP mode. This is achieved before single-step is checked because single-step is a post-execution exception. When the CPU leaves sleep mode, the single-step condition causes a single-step handler to be launched. The single-step condition takes priority over the asserted interrupt. This is consistent with the execution model described in *Section 16.16*. An alternative arrangement would be to accept the asserted interrupt first. This would be less satisfactory because a single-step event would be lost.

If the wake-up event is an NMI or a CPU reset, the behavior is slightly different because the NMI or CPU reset handler cannot be blocked. Let the NMI or CPU reset handler contain instructions I_l, I_m and I_n but assume that it does not attempt to RTE.

The sequence of instructions that will be executed is:

1 Previous instructions are executed such that SR.STEP is now set

2 Execute I_a

Launch DEBUGSS with SPC=I_b: execute I_x, execute I_y, execute I_z (RTE to I_b)

3 Execute I_b and enter sleep mode

After some cycles, an NMI or CPU reset causes exit from sleep mode

Launch DEBUGSS with SPC=I_c and PC=I_x

4 NMI or manual reset causes handler launch with SPC=I_x; execute I_l

5 Execute I_m

6 Execute I_n

7 Subsequent instructions are executed

Note that a single-step exception is launched at I_x with SPC set to I_c. However, before I_x can be executed, the NMI or CPU reset condition causes another launch with PC at I_l and SPC set to I_x. No instructions from the single-step handler are executed, but the transitional flow of control to the single-step handler is visible through the value of SPC delivered to the NMI or CPU reset handler.

Note that a CPU reset launch automatically clears SR.STEP while an NMI launch preserves SR.STEP. In this case, however, the earlier launch of the single-step handler causes SR.STEP to be cleared, and thus an NMI handler would not itself be single-stepped.

16.17 Interaction between debugger and target

The CPU's debug support consists of additional state and mechanism provided specifically for the debugger. This support can be protected from user mode access, but is fully accessible to privileged mode code. Target software should use the debug support in a way that is compatible with its debug requirements.

16.17.1 External debugger

The CPU architecture is arranged so that an external debugger can attach to the CPU and debug the target system. The external debugger and the target software can be highly decoupled allowing the debugging of arbitrary target software. When using an external debugger, the following conventions are recommended:

1. Target software should provide handlers using RESVEC to catch and handle debug exceptions. When DBRMODE is 0, external debugging features are not enabled and debug events will be vectored through RESVEC to target software.
2. DBRVEC and DBRMODE (see *Section 16.6.4: Handler addresses on page 231*) should be reserved for the debugger. They should not be accessed by target software. An external debugger can program DBRVEC and set DBRMODE to allow debug events to be handled by the external debugger. If the external debugger chooses not to handle a particular event, it can pass the handling of that event onto target software by emulating handler launch through RESVEC.
3. Target software should not access PSSR, PEXPEVT, PSPC or DCR (see *Section 9.3.2: Control register instructions on page 165*). This state should be reserved for the debugger.These control registers are used when handling debug events. Their value could be modified non-deterministically from the point of view of privileged mode target software.
4. Boot-strap code is the initialization code that executes after a reset. The debugger can arrange for the boot-strap to be debugged from the very first instruction executed after reset. For example, a debugger can use debug features provided by the system architecture to arrange for SR.STEP and/or SR.WATCH to be set. This allows single-stepping and/or watch-points to be enabled immediately after reset but before execution of the first instruction of the target program.

   Boot-strap code must be aware that SR.STEP and SR.WATCH can be modified from their reset value by the debugger before execution of the first instruction. Typically, a boot-strap will initialize architectural state (e.g. control and configuration registers), and then execute RTE. It is important that SR.STEP and SR.WATCH are preserved through this RTE so that single-stepping and watch-points are not inadvertently disabled. One way to achieve this is for the boot-strap to read SR, modify only those fields that the boot-strap needs to change, write the new value to SSR, and then use RTE.

5 In general, target software should preserve the current values of SR.STEP and SR.WATCH. This allows the debugger to control the single-stepping and watch-pointing properties of target software. For an operating system supporting multiple threads of execution, the operating system should preserve the values of SR.STEP and SR.WATCH independently for each thread to allow the debugger to control these features on a per-thread basis. Thus, SR.STEP and SR.WATCH should be considered as part of the thread context, and accordingly saved and restored at thread context switches.

6 The launch sequence for all debug exceptions causes EXPEVT to be set with the event code associated with that debug event. Additionally, for DEBUGOA and DEBUGIV events, TEA is set with the matched operand address or the address of the matched instruction. For non-panic events, it is not possible for the debugger to preserve EXPEVT and TEA across all debug handlers. Target software should not rely upon the values in EXPEVT and TEA when SR.BL is 0. Note that these registers can be preserved for panic events.

7 For non-panic debug events, the previous values of SPC and SSR are lost during the debug handler launch. It is not possible for the debugger to preserve SPC and SSR in these cases. Thus, target software should not rely upon these values when SR.BL is 0, since their values could be modified non-deterministically by debug software. It is not sufficient for target software to raise SR.IMASK to its highest level (15). Although this will mask external interrupts, it will not mask debug exceptions and debug interrupts. The value of SR.BL should be 1 across any code sequence that depends on the values of SPC and SSR being retained. Debug exceptions will then use the panic mechanism, preserving SPC and SSR. Note that all event handler launches automatically set SR.BL to 1.

Note that these conventions are recommendations and not mandatory.

16.17.2 Other debug arrangements

The statements in *Section 16.17.1* assume the use of an external debugger. There are alternative environments where software can choose to exploit the core debug features in other ways. For example:

- There could be a requirement for the target to prevent or hinder debug activities.
- There could be no requirement for the target to support any debug mechanism.
- The debugger could be integrated into the target software.

- The target could support both internal and external debug.

These arrangements could require different approaches and different software conventions to those described in *Section 16.17.1*.

16.18 Event handling and USR

Event handling code must take special care with the USR control register (see *Section 15.2.11: USR on page 219*). USR can be used to optimize context switch as described in *Section 2.8: Register subsets on page 25*. The key architectural property of USR is that when an instruction with a modifiable destination register is executed, the relevant bit of USR will be set.

The following approach is required for maximal preservation of USR across an event handling sequence:

- Following event launch, USR must be saved into a general-purpose register using GETCON before any instruction is executed that has a destination register.
- Before recovery from an event, USR must be restored from a general-purpose register using PUTCON. There must be no other instructions with a destination register between this PUTCON and the recovery RTE.

To implement these save and restore sequences, it is necessary to reserve a general-purpose register for the use of event handlers.

Alternatively, software can choose to preserve only a subset of USR. For example, if the event handler is allowed to modify one bit of USR.GPRS, then 8 different general-purpose registers can be written to in the launch and recovery sequences without further loss of information in USR.

The architecture leaves considerable flexibility in the implementation of the dirty bits. An implementation is allowed to set dirty bits under other circumstances in addition to those guaranteed by the architecture. Therefore, the degree to which USR can be preserved across an event handling sequence is a property of the implementation and not guaranteed by the architecture.

SuperH

17 Memory management

17.1 Introduction

The architecture provides a memory management unit (MMU). The MMU is present in all implementations, though the functionality of the MMU is scalable. This allows the memory management features supported by an implementation to be tailored to its requirements.

The main features that are provided by the MMU are:

- Disable/enable: a mechanism to allow the MMU to be disabled (for boot-strap) and enabled (for program execution).
- Protection: a mechanism to associate protection information with address ranges to allow those address ranges to be protected against inappropriate access.
- Cache control: a mechanism to associate cache behavior information with address ranges to allow control of the cache for those address ranges.
- Effective address space: all memory accesses made by executing instructions on the CPU are to addresses in effective address space.
- Physical address space: all memory accesses made by the CPU to the memory system are to addresses in physical address space.
- Translation (optional): if translation is not supported, then effective addresses are turned into physical addresses by an identity mapping. If translation is supported, then the mapping of effective addresses into physical addresses is programmable.

Some MMU properties are managed using MMU configuration registers. The organization of these registers is highly implementation dependent and is not described in this document.

17.2 Scalability

The MMU allows the following parameters to be scaled/varied between different implementations:

- Number of implemented bits in effective addresses.
- Number of implemented bits in physical addresses.
- Page sizes: number of page sizes, and their actual sizes.
- Caching: number of supported cache behaviors.
- Translation: supported or not supported.
- Number of effective address spaces.
- Organization and size of the translation description.

17.3 MMU enable and disable

After power-on reset or CPU reset, the CPU starts executing with the MMU disabled. The enable/disable state of the MMU can be accessed through the SR control register. The MMU can be enabled or disabled using the RTE instruction.

17.4 Address space

All implementations support effective addresses and physical addresses. The mapping between these depends on whether the implementation supports translation.

17.4.1 Physical addresses

The CPU core interacts with the physical memory system using physical addresses. There is a single physical address space. The contents of the physical memory map are determined by the system architecture not by the CPU core.

The total physical address space contains 2^{64} bytes. Physical addresses are unsigned and therefore vary in the range $[0, 2^{64})$.

Implementations do not necessarily implement all of the physical address space; the amount provided can vary between implementations. The number of bits in the implemented physical address, nphys, will be in the range [32, 64].

These implemented bits are always the least significant bits of the physical address. The implemented subset of the 64-bit total physical address space has the upper (64-nphys) bits of the physical address set to the same value as bit number (nphys-1).

The physical address space is depicted in *Figure 64*.

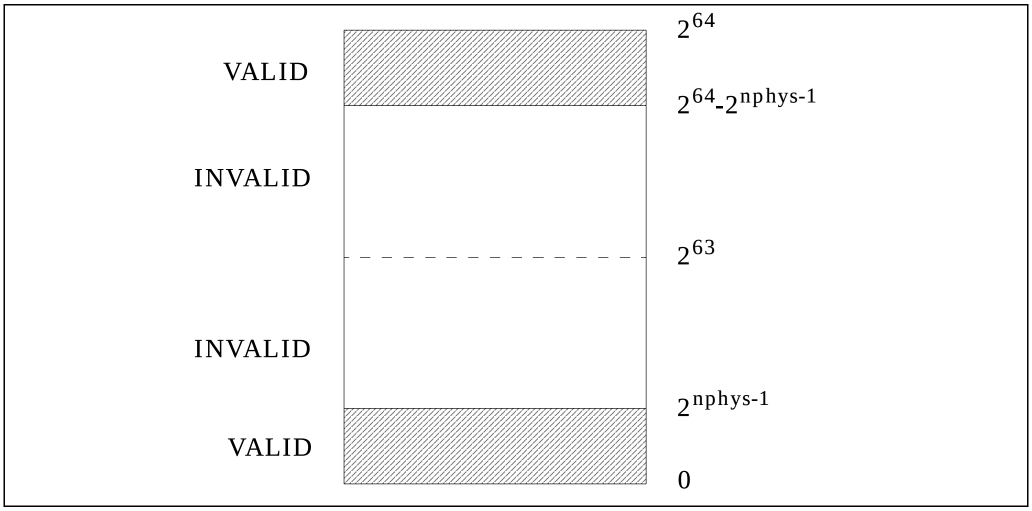


Figure 64: Physical address space for any valid nphys

Essentially the unimplemented bits follow the value of the most significant implemented bit. The implemented physical address space occupies physical addresses in the range [0, $2^{nphys-1}$) and the range [$2^{64}-2^{nphys-1}$, 2^{64}). In the case where nphys is 64, this collapses to a single range [0, 2^{64}).

17.4.2 Effective addresses

All memory accesses made by the CPU are characterized by an effective address and a data width. The total effective address space is 64 bits since the effective address computation in load/store instructions is performed to 64-bit precision.

The organization of the 64-bit effective address space is analogous to that of the physical address space. The total effective address space contains 2^{64} bytes. Effective addresses are unsigned and therefore vary in the range [0, 2^{64}). The total effective address space is flat.

Implementations do not necessarily implement all of the effective address space; the amount provided can vary between implementations. The number of bits in the implemented effective address is neff. If the implementation does not support translation then neff has the same value as nphys. If the implementation does support translation, then neff will be in the range [nphys, 64]. This means that the implemented effective address space is always sufficient to map all of the implemented physical address space.

These implemented bits are always the least significant bits of the effective address. The implemented subset of the 64-bit total effective address space has the upper (64-neff) bits of the effective address set to the same value as bit number (neff-1).

The effective address space is depicted in *Figure 65*.

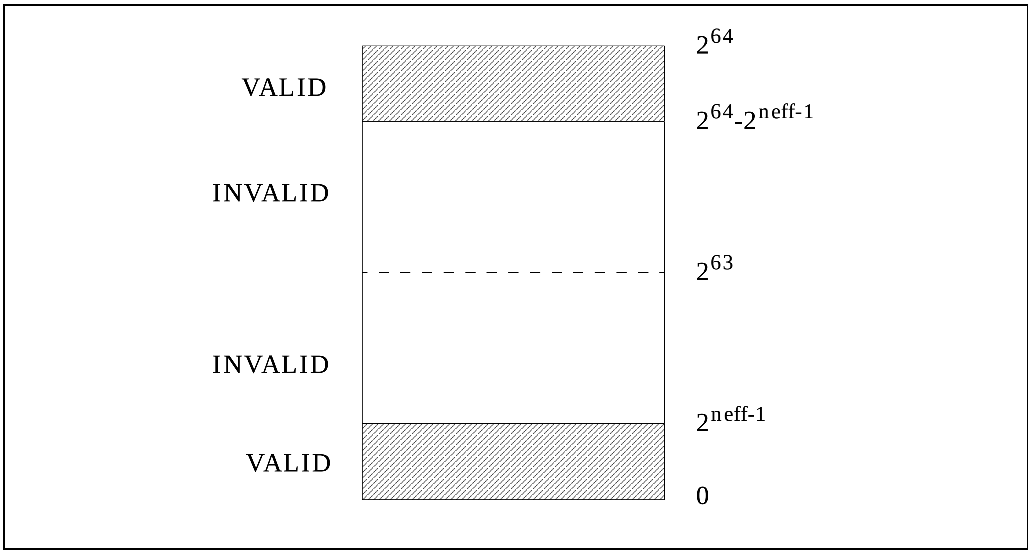


Figure 65: Effective address space for any valid neff

Essentially the unimplemented bits follow the value of the most significant implemented bit. The implemented effective address space occupies effective addresses in the range $[0, 2^{neff-1})$ and the range $[2^{64}-2^{neff-1}, 2^{64})$. In the case where neff is 64, this collapses to a single range $[0, 2^{64})$.

17.4.3 Virtual addresses

When the MMU is enabled and the MMU supports translation, conventional virtual address space can be supported. In this case, the term ‘virtual address’ is interchangeable with ‘effective address’.

However, if the MMU is disabled or if the MMU does not support translation, conventional virtual address space is not supported. In this case, it would be misleading to interchange the term ‘virtual address’ with the term ‘effective address’.

For this reason, this architecture is described in terms of effective addresses rather than virtual addresses.

17.4.4 Mapping from effective to physical addresses

The mappings from effective addresses to physical addresses are out-lined below. When the MMU is disabled, the mapping algorithm is common to all implementations. When the MMU is enabled, the mapping algorithm depends on whether the implementation supports translation.

When the CPU makes an access to an effective address, the mapping is achieved as follows:

1 The effective address is checked for validity. Validity checking increases compatibility between implementations with varying amounts of implemented effective address space. If neff is 64, then the effective address is always valid. Otherwise, if the effective address is in the range [2^{neff-1}, 2^{64} - 2^{neff-1}) then it is invalid and an appropriate exception is raised. The exceptions that are used for addresses outside the implemented effective address space are IADDERR for instruction addresses, RADDERR for loads and WADDERR for writes. Note that these exceptions are also used for misaligned access. If the effective address is valid, the mapping continues.

2 If the MMU is disabled, the effective address is converted directly into a physical address without translation. This mapping is described in *Section 17.6: Behavior when the MMU is disabled on page 277*.

3 If the MMU is enabled and the MMU does not support translation, the effective address is converted directly into a physical address without translation. This mapping is described in *Section 17.7.5: Effective address mapping without translation on page 286*. Although there is no address translation, various properties can be associated with the access.

4 If the MMU is enabled and the MMU does support translation, the effective address is converted into a physical address by a translation process. The translation mechanism supports multiple effective address spaces. Each effective address space is typically associated with a different process. The effective address spaces are distinguished by an address space identifier (ASID). This mapping is described in *Section 17.7.6: Effective address mapping with translation on page 287*. The current value of ASID is specified in SR.ASID (see *Section 15.2.1: SR on page 210*) and can only be changed using the RTE instruction.

17.5 Pages

The granularity for associating attributes with address space is the page.

Multiple page sizes can be supported, though the actual number of page sizes and their sizes are implementation defined. Any or all of the different page sizes can be in use at the same time.

Page sizes are always a power-of-2, 2^n, where n varies in the range [12, nphys]. The smallest possible page size is 4 kbytes, and the largest possible page size exactly fills the physical address space. A page in memory always starts at an address which is aligned to its page size.

Physical address space is partitioned into pages. For a page size, 2^n, bits 0 to n-1 of the physical address represent the byte-index within the page, and bits n to nphys-1 represent the physical page number (PPN).

Effective address space is also partitioned into pages. Translation information, if supported, is associated with each effective page. For a page size, 2^n, bits 0 to n-1 of the effective address represent the byte-index within the page, and bits n to neff-1 represent the effective page number (EPN).

Each memory access made by the instruction stream is fully contained within an 8-byte grain of memory aligned to an 8-byte boundary. This means that no accesses straddle a page boundary. Every access is fully contained within a single page.

17.6 Behavior when the MMU is disabled

After a power-on reset, a CPU reset or a panic, the CPU executes code with the MMU disabled. Execution of code with the MMU disabled is well behaved regardless of the state of the MMU configuration registers. This is important because MMU implementations can contain many programmable fields and these fields have an architecturally-undefined value after power-on reset.

The intention is that the amount of code that executes with the MMU disabled is very small. This code, typically called a boot-strap, needs to program the MMU with an appropriate memory management configuration and then enable the MMU. The details of the configuration depend upon the memory management features provided by the implementation. The initial MMU configuration can be very simple depending on the implementation and the required mappings.

The speed of execution of code when the MMU is disabled is not critically important. This is because one of the first actions of the boot-strap code will be to configure the MMU and enable it. This can be achieved with a relatively small number of instructions. This means that the execution model for code when the MMU is disabled can be very simple.

When code executes with the MMU disabled:

- Effective addresses are mapped directly to physical addresses. This mapping is essentially an identity translation.

  However, in the case where the implementation supports more effective address space than physical address space (that is, neff > nphys), the physical address space appears replicated throughout the effective address space. The effective address (EA) is mapped to a physical address (PA) by sign-extending EA from bit number nphys-1. Thus, any bits of PA in the range [nphys, 64) will be copies of bit number nphys-1 of PA. This mapping is exactly an identity translation when neff and nphys are identical.
- There is no protection mechanism.
- All data accesses, including swap accesses, are implemented as though they were device accesses. The data cache is frozen and bypassed. The precise amount of data is transferred. There is no speculative data access.
- Instruction fetches are not cached. The instruction cache is frozen and bypassed. Additionally, the amount of speculative instruction fetch is restricted to avoid prefetches from device areas of physical memory.

Properties of speculative memory access are described in *Section 18.11: Speculative memory accesses on page 311*.

Since accesses are not cached while the MMU is enabled, optimal performance cannot be achieved. It is strongly recommended that the MMU is configured and enabled as soon as possible after reset.

17.7 Behavior when the MMU is enabled

When the MMU is enabled, the mappings from effective addresses to physical addresses are described using page table entries (PTE). Each page table entry consists of two configuration registers which are called the high PTE (PTEH) and the low PTE (PTEL). The distinction between 'high' and 'low' is for naming convenience only. These specify the properties of that page in effective and physical address space. Page table entries are held in an array to allow multiple pages to be described.

A PTE array is also called a translation lookaside buffer (TLB).

The following sections describe the organization of the PTE arrays, the contents of the PTE configuration registers, the mapping mechanisms and implementation options.

17.7.1 PTE array organization

There are two possible organizations of the page table entry arrays: unified and split:

- A unified organization consists of a single array of page table entries. Each entry controls the behavior of both data and instruction accesses to the described page. The number of entries in the array is implementation defined and is represented here by u.

  The configuration registers in the unified array are called:

  - MMUDR[n].PTEH
  - MMUDR[n].PTEL

  where n varies in the range [0, u).

- A split organization consists of two arrays of page table entries. An entry in the data register array controls the behavior of data accesses to the described page, whereas an entry in the instruction register array controls the behavior of instruction accesses to the described page. The number of entries in these arrays is implementation defined and is represented here by d for the data register array and i for the instruction register array.

The configuration registers in the data array are called:

- MMUDR[n].PTEH
- MMUDR[n].PTEL

where n varies in the range [0, d). The configuration registers in the instruction array are called:

- MMUIR[n].PTEH
- MMUIR[n].PTEL

where n varies in the range [0, i).

All entries in a PTE array are equivalent. The PTE arrays are fully associative. Each entry can hold information for any effective to physical address mapping.

17.7.2 MMU configuration registers

The MMU configuration registers are summarized in the following diagrams. Further details of configuration register layout are implementation dependent. Each configuration register is 64 bits wide. The annotations above each register denote field widths, while the annotations below denote bit numbers.

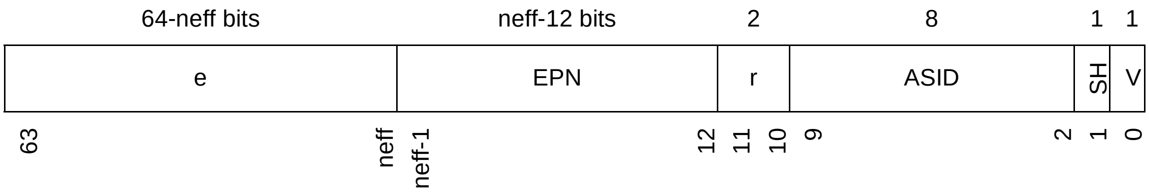


Figure 66: PTEH configuration register

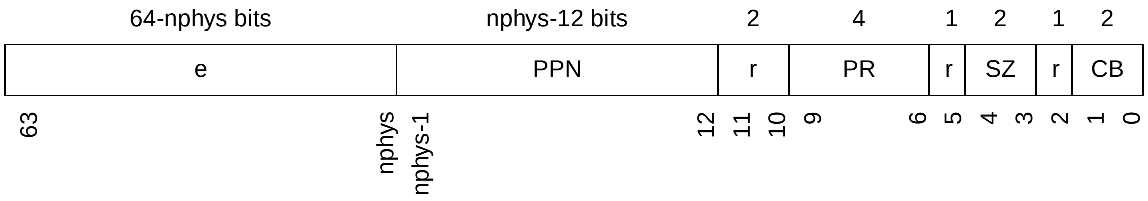


Figure 67: PTEL configuration register

The ‘r’ field indicates bits that are reserved for future expansion of the architecture. Software should ignore the value read from reserved bits, and write these bits as 0.

Possible future uses of the ‘r’ fields include:

- Extension of ASID to increase the number of supported address spaces.
- Extension of CB to increase the number of supported cache behaviors.
- Extension of SZ to increase the number of supported page sizes.
- Extension of PR to increase the number of supported protection attributes.

The ‘e’ field indicates bits reserved for future expansion of the address space using a sign-extended convention. Software should ignore the value read from these bits. Software should write a sign-extension of the highest implemented bit into expansion bits. This approach is necessary if software is to be executed on a future implementation with more implemented address space.

Note that the above information does not require future architecture versions to use these options, nor does it constrain future architecture versions to just these options.

Table 162 indicates where the descriptions of these registers and fields can be found.

State	Description	Behavior
PTEH.EPN	Effective page number	See *Effective page number (PTEH.EPN) on page 286*
PTEH.ASID	Address space identifier	See *Address space identifier (PTEH.ASID) on page 286*
PTEH.SH	Shared page flag	See *Shared page (PTEH.SH) on page 285*
PTEH.V	PTE enable	See *Enable (PTEH.V) on page 282*
PTEL.PPN	Physical page number	See *Physical page number (PTEL.PPN) on page 285*
PTEL.PR	Protection information	See *Protection (PTEL.PR) on page 284*
PTEL.SZ	Page size	See *Page size (PTEL.SZ) on page 282*
PTEL.CB	Cache behavior information	See *Cache behavior (PTEL.CB) on page 282*

Table 162: PTEH and PTEL fields

The PTEH and PTEL configuration registers are replicated to provide an array of PTEs that describe the available mappings from effective to physical addresses.

17.7.3 Implementation options

In addition to the variation possible in the PTE array organization, the MMU architecture supports three different implementations of the PTE state. These are:

1. Translation not supported, implemented PTE fields are read-only: this variant gives a set of hard-wired non-translated mappings, and results in a very simple implementation. The PTE look-up can be implemented by decoding bits from the effective address, rather than by an associative look-up into a PTE array. This variant only supports systems with very simple MMU requirements.
2. Translation not supported, implemented PTE fields are read-write: this variant gives programmable control of protection and caching at the page level, but without support for translation. This variant supports systems that require protection without the cost of translation.
3. Translation supported, implemented PTE fields are read-write: this variant is fully featured and can support standard virtual memory.

A summary of the semantics of each PTE field is given in *Table 163* to show how the behavior varies for these different implementations.

PTE field	Translation not supported		Translation supported
	Implemented PTE fields are read-only	Implemented PTE fields are read-write	Implemented PTE fields are read-write
PTEH.EPN	RES	RES	RW
PTEH.ASID	RES	RES	RW
PTEL.PPN	RO	RW	RW
PTEL.PR	RO	RW	RW
PTEL.CB	RO	RW	RW
PTEH.SH	RO	RW	RW
PTEL.SZ	RO	RW	RW
PTEH.V	RO	RW	RW

Table 163: PTE implementation options

17.7.4 PTE contents

This section describes the fields within the PTE configuration registers. Some fields are only provided on implementations that support translation. The behavior of some fields depends on whether the PTE array organization is unified or split.

Enable (PTEH.V)

An enable bit is provided to control whether this PTE is enabled or disabled. This allows software to disable unused PTEs, and to ensure that PTEs are disabled while they are programmed. The enable field (PTEH.V) is described in *Table 164*.

Name of field	Size in bits	Behavior
PTEH.V	1	If 0: the PTE is disabled If 1: the PTE is enabled

Table 164: V field

Page size (PTEL.SZ)

The number of supported page sizes, npage, can vary between implementations, though every implementation must provide at least 1 page size. The sizes of the supported pages are also implementation defined. The page size field (PTEL.SZ) is described in *Table 165*.

Name of field	Size in bits	Behavior
PTEL.SZ	2	Distinguishes up to 4 supported page sizes.

Table 165: Page size field

Future versions of the architecture reserve the option to extend PTEL.SZ to support more page sizes (see *Section 17.7.2: MMU configuration registers on page 279*).

Cache behavior (PTEL.CB)

The implementation can optionally provide instruction and data caches. Different cache behaviors can be selected to allow the behavior of the cache to be specified at the page level. If caches are not supported, then the cache behavior field should be set to uncached.

The different cache behaviors are distinguished using the cache behavior field (PTEL.CB). Cache behavior is a property of the physical page, and must be used consistently at the granularity of the smallest page size supported by the architecture (4 kbytes). The architecture allows multiple mappings to share the same physical page with different cache behaviors, providing that they are accessed mutually exclusively at this granularity. Thus, any particular 4 kbyte physical page must be accessed consistently with one cache behavior. The actions required to change the cache behavior of a particular physical page are described in *Section 17.8.3: Cache coherency when changing the page table on page 291*. If a physical page is not accessed with a consistent cache behavior, the behavior of memory accesses to that page will be unpredictable and cache paradox conditions can result (see *Section 18.9: Cache paradoxes on page 308*).

The available instruction cache behaviors are:

- Cached instruction fetches.
- Uncached instruction fetches.

The available data cache behaviors are:

- Cached accesses with write-back behavior.
- Cached accesses with write-through behavior.
- Device accesses (these are uncached and the exact amount of data is accessed).
- Uncached accesses (these are uncached but the accesses can be implemented more efficiently than is permitted for device pages).

Future versions of the architecture reserve the option to extend PTEL.CB to support more cache behaviors (see *Section 17.7.2: MMU configuration registers on page 279*).

The cache behavior field is described in *Table 166*. If a RESERVED setting is used, then the behavior is architecturally undefined.

Name of field and size	PTEL.CB value	Behavior with a unified PTE array		Behavior with a split PTE array	
		Instruction fetch	Data access	Instruction fetch	Data access
PTEL.CB 2 bits	0x0	uncachable	uncachable	uncachable	uncachable
	0x1	uncachable	device	RESERVED	device
	0x2	cachable	cachable, write-back	cachable	cachable, write-back
	0x3	cachable	cachable, write-through	RESERVED	cachable, write-through

Table 166: Cache behavior field

Protection (PTEL.PR)

Accesses are checked for various kinds of protection violation. Protection violation causes an appropriate exception to be raised. Protection is a property of the effective page. There is no requirement for mappings that share the same physical page to use the same protection attributes.

Each PTE has a protection field (PTEL.PR) containing the following bits:

1. PTEL.PRU: when set the page is accessible to user and privileged mode, otherwise it is accessible to just privileged mode.
2. PTEL.PRW: when set the page is writable, otherwise non-writable.
3. PTEL.PRR: when set the page is readable, otherwise non-readable.
4. PTEL.PRX: when set the page is executable, otherwise non-executable.

Future versions of the architecture reserve the option to extend PTEL.PR to support more protection attributes (see *Section 17.7.2: MMU configuration registers on page 279*).

Permission is granted to privileged mode for an access if the appropriate access permission is given regardless of the value of PTEL.PRU. Permission is granted to user mode for an access if the appropriate access permission is given and PTEL.PRU is set. Prohibited accesses raise an appropriate exception.

The protection field is described in *Table 167*.If a RESERVED setting is used, then the behavior is architecturally undefined.

Name of field and size	Bit number of PTEL.PR	Value of bit	Interpretation with a unified PTE array	Interpretation with a split PTE array	
				Instruction side	Data side
PTEL.PR 4 bits	0 (R)	0	not readable	BIT MUST BE 0	not readable
		1	readable	RESERVED	readable
	1 (X)	0	not executable	not executable	BIT MUST BE 0
		1	executable	executable	RESERVED
	2 (W)	0	not writable	BIT MUST BE 0	not writable
		1	writable	RESERVED	writable
	3 (U)	0	privileged only	privileged only	privileged only
		1	privileged and user accessible	privileged and user accessible	privileged and user accessible

Table 167: Protection field

Physical page number (PTEL.PPN)

For a page size of 2^n bytes there are (nphys-n) bits in the PPN. The PTEL.PPN field contains sufficient bits to support the smallest page size allowed by the architecture (4 kbytes). Thus, PTEL.PPN contains (nphys-12) bits. Where the actual page size is greater than this smallest page size, the PPN must be stored in the most significant bits of the PTEL.PPN field and the remaining least significant bits of PTEL.PPN must be cleared.

Shared page (PTEH.SH)

This field is provided only on implementations that support translation. The shared page field (PTEH.SH) is used to control sharing of pages between different ASID values. It is used in the effective address look-up mechanism described in *Section 17.7.6: Effective address mapping with translation on page 287*.

The PTEH.SH field is described in *Table 168*.

Name of field	Size in bits	Behavior
PTEH.SH	1	If 0: the page is not shared If 1: the page is shared

Table 168: SH field

Address space identifier (PTEH.ASID)

This field is provided only on implementations that support translation. The PTEH.ASID field is used to distinguish different effective address spaces. It is used in the effective address look-up mechanism described in *Section 17.7.6: Effective address mapping with translation on page 287*. The value of PTEH.ASID is irrelevant for a shared page.

The number of provided effective address spaces can vary between implementations. The PTEH.ASID field contains 8 bits, but the number of these bits that are implemented is implementation dependent. If the number of implemented bits is nasids, then nasids is in the range [0,8] and this field can be programmed with values in the range $[0, 2^{nasids})$, while any values in the range $[2^{nasids}, 2^8)$ are reserved.

Future versions of the architecture reserve the option to extend PTEH.ASID to support more address spaces (see *Section 17.7.2: MMU configuration registers on page 279*).

The PTEH.ASID field is described in *Table 169*.

Name of field	Size in bits	Behavior
PTEH.ASID	8	Distinguishes up to 256 different effective address spaces

Table 169: ASID field

Effective page number (PTEH.EPN)

This field is provided only on implementations that support translation.

For a page size of 2^n bytes there are (neff-n) bits in the EPN. The PTEH.EPN field contains sufficient bits to support the smallest page size allowed by the architecture (4 kbytes). Thus, PTEH.EPN contains (neff-12) bits. Where the actual page size is greater than this smallest page size, the EPN must be stored in the most significant bits of the PTEH.EPN field and the remaining least significant bits of PTEH.EPN must be cleared.

17.7.5 Effective address mapping without translation

This section describes the effective address mapping procedure for implementations that do not support translation. Implementations with translation follow the procedure described in *Section 17.7.6: Effective address mapping with translation on page 287*.

The values of neff and nphys are identical for an implementation that does not support translation. Effective addresses are mapped directly to physical addresses. This mapping is an identity translation: the physical address is identical to the effective address. An identity mapping is sufficient since the range of valid effective addresses exactly matches the range of physical addresses.

This physical address is then used to perform an associative look-up in the appropriate PTE array. A match is found if the physical page described by a PTE contains the physical address of the access.

If a match is found, the look-up determines the protection and cache attributes to be used for that access. If a match is not found, then an exception is raised to indicate an instruction miss (ITLBMISS) or data miss (RTLBMISS or WTLBMISS).

The content of the PTE arrays must be arranged such that there is, at most, one PTE that describes the mapping of any physical address. If there are multiple mappings for any physical address, then the behavior is architecturally undefined.

17.7.6 Effective address mapping with translation

An implementation can optionally support translation. The required mechanisms are described here. None of this mechanism is needed for implementations that do not support translation.

Translation gives flexible control over the mappings from effective addresses into physical addresses. Standard virtual memory can be supported by using effective address space, the translation mechanism and appropriate software. In this case, the virtual memory map is determined entirely by software not by the CPU architecture.

This effective address mapping is achieved as follows. The effective address of the access and the ASID of the current process are used to perform an associative look-up into the appropriate PTE array. The following checks are made against each PTE:

- An effective address match is found if the EPN of the effective address of the access matches the PTEH.EPN field. Note that the bits of the effective address used in this comparison depend on the page size of that PTE. For a page of size 2^n bytes, bits n to neff-1 inclusive of the effective address are compared.
- An ASID match is found if PTEH.SH is 1, or if the ASID of the current process matches the PTEH.ASID field. The PTEH.SH allows pages to be shared across all processes regardless of ASID.

A PTE match requires an effective address match and an ASID match in the same PTE.

If a PTE match is found, the look-up determines the attributes (physical page number, protection and cache attributes) to be used for that access. The translation from effective address to physical address is achieved in the conventional way by substituting the physical page number for the effective page number. Thus, the byte-index within the page is retained, and the EPN is replaced by the PPN. This process is illustrated in *Figure 68*.

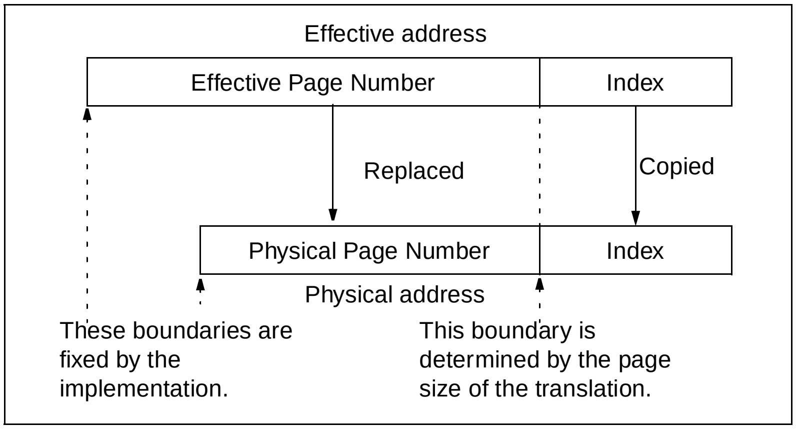

Effective address
Effective Page Number
Index
Replaced
Copied
Physical Page Number
Index
Physical address
These boundaries are fixed by the implementation.
This boundary is determined by the page size of the translation.

Figure 68: Effective to physical mapping

If a PTE match is not found, then an exception is raised to indicate an instruction miss (ITLBMISS) or data miss (RTLBMISS or WTLBMISS). This exception can be used to cause software refill of the appropriate PTE array, and to detect accesses to invalid addresses. PTE refill is performed completely in software; there is no hardware page-table walking.

There must be, at most, one PTE that describes the mapping of any effective address in any effective address space. If there are multiple mappings present for any effective address and ASID combination, then the behavior is architecturally undefined.

17.7.7 Mappings required to execute an instruction

The number of mappings required for an instruction to execute is a property of the instruction set architecture. Note that instructions and memory accesses never straddle page boundaries.

Each SHmedia instruction requires:

1. A mapping for its instruction bytes.
2. If it is an instruction that accesses memory, a mapping for its data.

Each SHcompact instruction requires:

1. A mapping for its instruction bytes.
2. If it is an instruction that accesses memory, one or two mappings for its data. Some SHcompact instructions (specifically, MAC.W and MAC.L) read two separate memory locations. Mappings must be simultaneously available for both read accesses in order for such instructions to execute without a page miss exception.

In summary, any instruction can be executed using one mapping for its instruction bytes and at most two mappings for its data.

17.8 MMU and caches

This section describes the interaction between the memory management unit and the caches. This elaborates on information presented earlier in this chapter. Further information on these mechanisms can be found in the cache chapter (see *Chapter 18: Caches on page 297*).

This section is not relevant to implementations without caches.

17.8.1 Cache behavior when the MMU is disabled

When the MMU is disabled, all cache state is bypassed and frozen with respect to accesses. This behavior is provided regardless of whether the caches are themselves enabled. The cache enable flag only has an effect when the MMU is enabled. Thus, in the case that the MMU is disabled but the caches are enabled, the cache state is still bypassed and frozen.

Bypassing means that accesses do not see the state of any caches; essentially, accesses always miss a bypassed cache. Freezing means that accesses do not modify the state of any cache. In effect, accesses proceed as if the cache were not present.

Cache coherency instructions and the cache configuration mechanisms still operate on the cache state, and will access the cache as usual. This provides software with a means to manage the cache state regardless of whether the MMU is enabled or disabled.

There are a number of advantages to this arrangement:

- The behavior of the cache when the MMU is disabled is fully specified, allowing the well-behaved execution of instructions without encountering paradoxical cache situations.
- After a CPU reset, software can observe the complete state of the cache prior to the reset. This is important for post-mortem debugging.
- In normal operation the MMU is enabled. It is possible to arrange for the MMU to be disabled, instructions to be executed without translation, and the MMU to be re-enabled without affecting the cache state. This behavior allows the system to support fully-decoupled, interactive debugging. Essentially, a debugger can arrange to run its own code, with the MMU disabled, without affecting the functional behavior of the target system.

17.8.2 Cache behavior when the MMU is enabled

When the MMU is enabled, the behavior of the caches can be programmed by software. The cache behavior is specified at the page level using PTEL.CB, but this setting can be globally over-ridden by the cache configuration. The semantics of cache behavior are described in the cache chapter (see *Chapter 18: Caches on page 297*).

When the MMU is enabled, software is responsible for guaranteeing that the caches are used in a safe way. In particular, software must ensure that cache paradoxes are avoided. A cache paradox occurs when a memory access finds that the current cache state is inconsistent with the required cache behavior. An example is a device access which finds that the accessed data is in the cache; this situation is inconsistent with device semantics.

Software conventions must be used to prevent such situations. It is necessary to ensure that all mappings that share the same physical page have the same cache behavior, otherwise the behavior of memory accesses to that page will be unpredictable. A more detailed description of the necessary constraints can be found in *Section 18.8.3: Cache behavior on page 305*. Particular care is required when changing page table contents; this is described in *Section 17.8.3: Cache coherency when changing the page table on page 291*.

17.8.3 Cache coherency when changing the page table

For implementations that have read-write PTE fields, software is able to change the contents of a PTE. The MMU architecture places a usage model on page table updates to allow a wide variety of implementations. This model requires software to honor certain constraints when changing the contents of a page mapping.

The MMU architecture uses the model that the entries in the PTE arrays (the hard PTEs) are a subset of a larger set of notional PTE values maintained in some way by software (the soft PTEs). Software is given complete freedom as to how the soft PTEs are managed. For example, they can be stored in a memory-held PTE data structure, they can be calculated dynamically, or whatever.

The MMU is informed of the existence of a soft PTE at the point where that PTE is loaded into a hard PTE and enabled. While the MMU is informed of the existence of a soft PTE, the MMU can (optionally) cache the hard PTE into a cached PTE. The cached PTE allows the MMU to retain the state of the soft PTE even when the corresponding hard PTE has been reprogrammed. This property supports implementations that copy PTE information in the cache. This arrangement is called a virtual cache.

Under normal use, software will evict entries from hard PTEs and refill from soft PTEs as required by page misses. These evictions and refills do not generally require the state of the soft PTEs to be changed, and no special operations are required to keep the cached PTE state coherent.

When a soft PTE is modified then the cached PTE state must be made coherent by explicit software actions. This can be achieved by software by meeting the following two conditions (at the same time):

- There must be no enabled hard PTE corresponding to the soft PTE. This can be achieved by disabling the hard PTE, if any, which corresponds to that soft PTE.
- There must be no valid or dirty lines in any cache corresponding to effective addresses mapped by that soft PTE. This condition is automatically satisfied if the cache behavior of the PTE is device or uncached. If it is cached, the condition must be satisfied through an appropriate cache coherency operation.Cache coherency is described in *Chapter 18: Caches on page 297*.

When these two conditions are met, then the MMU is no longer aware of the existence of the soft PTE and the soft PTE can be safely modified.

The soft PTE identifies an effective page in the effective address space defined by PTEH.EPN, PTEH.ASID and PTEL.SZ. The following scenarios constitute modifications to the soft PTE:

1 The effective page is being demapped (that is, becomes no longer accessible).

2 The effective page is being remapped (that is, PTEL.PPN is being changed).

3 The sharability (PTEH.SH) of the effective page is being changed.

4 The cache behavior (PTEL.CB) of the effective page is being changed. Note that cache behavior is a property of the physical page, and must be used consistently at the granularity of the smallest page size supported by the architecture (see *Section : Cache behavior (PTEL.CB) on page 282*).

5 The protection properties (PTEL.PR) of the effective page are being changed such that one or more of the protection attributes has an increase in protection (that is, there are accesses to the old PTE which are no longer permitted to the new PTE). This occurs if *any* of the permission bits are changed from 1 to 0.

If none of the protection bits are changed so as to increase their protection (that is, each bit is either unchanged or is changed to allow more accesses), this does not count as a PTE modification. This concession allows software to avoid PTE coherency costs in some important cases.

For example, software can mark a clean page as non-writable, catch the initial write exception and then enable write permission in the hard PTE. Enabling write permission does not require PTE coherency. The architecture guarantees that if a protection violation is detected in the cached PTE, the implementation will refer to the hard PTE to determine whether an exception is to be launched.

17.8.4 Cache synonyms

The architecture allows the implementation to select whether cache look-ups are performed using index bits from the effective address or from the physical address (see *Section 18.6: Cache mapping on page 301*). If an implementation chooses to use index bits from the effective address, then it is possible for cache synonyms to occur depending on the size and organization of the cache.

Cache synonyms occur when data from the same physical location is present in two *different* sets in the cache[1]. For data that can be modified, cache synonyms can lead to incorrect memory behavior since there is no mechanism defined by the architecture to keep the modifications consistent between these different sets. Cache synonyms are a potential problem only when multiple translations (with different effective addresses) are used to map the same physical location.

1. Please see *Chapter 18: Caches on page 297* for further details of cache organization and behavior including definitions of cache blocks and sets.

The following sections describe how the architecture manages cache synonym problems.

The architecture also recognizes a related, but distinct, property of caches. This is the concept of cache aliases, which is described separately in *Section 18.10: Cache aliases on page 309*. Cache aliases occur only in caches when the tags in the cache are derived from the effective address rather than the physical address and where the cache look-up is completed using just the effective address without an address translation.

Such implementations must consider whether the same physical location can be present in two different cache blocks in the *same* set since this results in cache aliases. Like cache synonyms, cache aliases also lead to incorrect memory behavior and are a potential problem only when multiple translations (with different effective addresses) are used to map the same physical location.

The architecture chooses to use different names for 'cache synonyms' and 'cache aliases' because very different techniques are used in the architecture to solve these distinct problems. Cache synonyms are multiple copies of the same physical location occurring in different sets due to indexing into the cache using the effective address. Cache aliases are multiple copies of the same physical location occurring in the same set due to the use of effective address tags in the cache. The use of different names for these concepts leads to a clearer and simpler description. Please note that some other architectures use these terms interchangeably without making any such distinction.

17.8.5 Instruction cache synonyms

The architecture allows an implementation to use an instruction cache organization that leads to synonyms. Since there is no mechanism to write into the instruction cache, synonyms in the instruction cache will remain coherent with each other. Therefore, the architecture imposes no constraints on software for managing instruction cache synonyms.

However, for some applications it can be desirable to avoid instruction cache synonyms. For example, instruction cache synonyms cause instructions to be held in multiple places in the instruction cache which can lead to lower utilization of the instruction cache. Additionally, if a physical location has instruction cache synonyms then complete invalidation of that location in the instruction cache requires all of the synonyms to be invalidated.

If software chooses to avoid instruction cache synonyms then the constraints specified in *Section 17.8.7: Constraints to avoid cache synonyms on page 294* should also be applied to instruction translations. Note again that the architecture and implementations do not require these constraints for correct instruction cache operation.

17.8.6 Operand cache synonyms

Unlike the instruction cache, the operand cache supports writes and operand cache synonyms must be avoided. The constraints described in *Section 17.8.7: Constraints to avoid cache synonyms on page 294* are mandatory for correct operand cache operation.

17.8.7 Constraints to avoid cache synonyms

When translation is used, cache synonyms can be avoided by placing constraints on the values of PTEH.EPN and PTEL.PPN for cachable pages.

nsynbits

The constraint is specified by a parameter, nsynbits, which has an implementation-specific value. This parameter gives the number of least significant bits of PTEH.EPN and PTEL.PPN that can suffer from cache synonyms. These bits are called synonym bits.

Note that the smallest page size supported by the architecture is 4 kbytes, and thus both PTEH.EPN and PTEL.PPN do not include the least significant 12 bits of the address.

Cache synonyms will be avoided if PTE values for all cachable pages, regardless of their page size, are programmed such that the synonym bits have identical values in all PTEH.EPN instances that map the same PTEL.PPN. This constraint allows cache implementations to index into the cache using lower order bits from the effective address rather than the physical address. This implementation option is described further in the cache chapter (see *Chapter 18: Caches on page 297*).

Note that if the selected page size is $2^{12+\text{nsynbits}}$ bytes or larger, then the constraint is automatically honored due to page alignment.

nsynmax

The architecture specifies that nsynbits will be in the range [0, nsynmax] for all implementations. This means that bits of PTEH.EPN and PTEL.PPN above nsynmax never suffer from synonym problems.

The value of nsynmax is 4. For example, an implementation can require that cachable mappings using a 4 kbytes page are constrained by 4 synonym bits.

However, it is guaranteed that an implementation will not constrain mappings that use 64 kbytes page size or larger.

Software implications

It is highly recommended that software honors the stricter architecturally-defined nsynmax constraint, rather than the weaker implementation-specific nsynbits constraint. This guarantee allows software to arrange its memory mappings in a way that will be compatible with future implementations.

The following discussion assumes that software honors nsynmax. If software chooses to make use of the implementation-specific nsynbits then replace nsynmax with nsynbits in the following paragraphs.

To avoid cache synonyms, software has to arrange the memory mappings of cachable pages such that bits [0, nsynmax) of all PTEH.EPN instances that map the same PTEL.PPN are identical. This constraint applies regardless of the page sizes being used. If a particular PTEL.PPN is only mapped once then there is no constraint. However, if there are 2 or more mappings of a particular PTEL.PPN, then software must arrange the PTEH.EPN values to satisfy this constraint.

Where multiple page sizes are used to map the same PTEL.PPN, the larger page size can force a constraint onto the smaller page size due to the stricter alignment of the larger page.

For example, consider an implementation supporting 4 kbytes and 64 kbytes pages where a particular physical address is mapped by pages of both these sizes. Since the 64 kbytes page is aligned to 64 kbytes, the bits in [0,4) of PTEH.EPN and PTEL.PPN for the 64 kbytes page will all be zero. In order to satisfy the synonym constraint, this forces the 4 kbytes page to have the same value in bits [0,4) of its PTEH.EPN as bits [0,4) of its PTEL.PPN. The overall effect, in this example, is that there are no choices available when selecting the synonym bits in the smaller page. Effectively, the required constraint has become that the synonym bits in PTEH.EPN must match the synonym bits in PTEL.PPN for both of these pages.

The recommended approach is for software to only allow translations that are consistent with these constraints. If software chooses to employ translations that are inconsistent with these constraints, then great care is required. Specifically, software must enforce cache coherency to avoid cache synonyms. If software has made accesses through a particular translation, then any cache entries corresponding to that translation must be removed by an appropriate cache coherency operation, before accesses can be safely made through an inconsistent translation. Cache coherency is described in *Chapter 18: Caches on page 297*.

Caches 18

18.1 Overview

A cache is a high-performance associative memory used for holding frequently accessed data or instructions close to the CPU. Caches exploit predictable memory usage patterns that occur in many programs, for example spatial and temporal locality of access. A cache can delay, aggregate, eliminate and re-order memory accesses. These techniques enable high load/store performance even where memory latency is substantially higher than the CPU cycle time.

This chapter introduces the principles of cache operation. Cache instructions are described in *Section 6.6 Cache instructions on page 101*. Some cache properties are managed using cache configuration registers. The organization of these registers is highly implementation dependent and is not described in this document.

18.2 Cache architecture

This section describes the architectural properties and behavior of caches. However, many aspects are highly implementation dependent. This includes, for example, whether that implementation provides any caches, and if it does how those caches are organized and exactly how those caches behave.

Although caches can have a significant effect on performance, the presence of the cache is functionally transparent to most software. This is because caches do not generally affect the memory model when viewed from just an instruction stream. Software that manages the cache directly, however, is exposed to the implementation-dependent properties of the cache.

Some properties of the cache can be described by implementation-specific parameters. Software that manages the cache should be written in terms of these parameters and provide mechanisms to allow the parameters to be set appropriately for the target implementation. Ideally, these parameters should be configurable at load-time or run-time to allow binary-level compatibility between implementations with different cache organizations. However, in some circumstances it can be necessary to bind this information statically into programs.

Two mechanisms are provided for cache management:

- Cache prefetch, allocate and coherency instructions: these are available to user and privileged mode, and insulate software from most implementation-specific cache properties.
- Cache configuration registers: these can be accessed using the configuration space from privileged mode. This is a highly implementation-specific mechanism. Any software that uses this mechanism will require significant attention should it be ported to another implementation with a different cache organization.

The strong recommendation of the architecture is that cache configuration registers should be used sparingly by software. The cache prefetch, allocate and coherency instructions should be used instead where they can achieve the desired effect.

18.3 Cache organization

The CPU architecture defines the behavior for one level of caches. The organization of the caches is implementation specific. Each implementation will select a level one cache organization from the following alternatives defined by the architecture:

- No cache: all accesses are performed on memory without caching.
- Unified cache: data and instruction accesses pass through a single cache.
- Split cache: data and instruction accesses are treated separately. The following implementation-specific options are available:
  - Only an operand cache is provided. Data accesses pass through the operand cache, while instruction accesses are performed on memory without caching. The terms ‘operand cache’ and ‘data cache’ are interchangeable.
  - Only an instruction cache is provided. Instruction accesses pass through the instruction cache. Data accesses are performed on memory without caching.
  - Both an operand cache and an instruction cache are provided. Data access pass through the operand cache, while instruction accesses pass independently and separately through the instruction cache.

The choice of cache organization is independent of the PTE organization of the MMU (see *Section 17.7.1 PTE array organization on page 278*).

Where an implementation provides a separate instruction cache, no mechanisms are provided to write memory through that cache.

An implementation or an external memory system can provide more levels of caches. The CPU architecture does not define how any such caches behave or are controlled.

18.4 Cache block

The unit of allocation in the cache is the cache block. A cache block is used to hold a copy of the state of some memory block. The terms ‘cache block’ and ‘cache line’ are interchangeable.

A cache block consists of data and address information:

- The data is used to hold a copy of the memory block.
- The address information is used to provide additional information specific to the memory block (if any) that is currently being cached. The precise information is implementation specific, but generally it consists of the following parts:
  - A flag to indicate whether that cache block is in use (valid) or not in use (invalid).
  - A flag to indicate whether that cache block contains data has not yet been written-back to memory (dirty) or not (clean).
  - Information to identify the memory block in the address map.
  - Cache, access and replacement information for that cache block.

The number of bytes of data associated with a cache block is called the cache block size. The cache block size is nbytes where nbytes is a power-of-2. The value of nbytes is at least the register size of the architecture (8 bytes) and at most the smallest page size of the architecture (4 kilobytes). The actual value of nbytes is implementation specific. If an implementation provides separate instruction and operand caches, then the cache block size can be different for each cache.

A memory block is a contiguous block of memory bytes. The memory block size is nbytes. The physical and effective addresses of the memory block are exact multiples of nbytes (that is, cache block size aligned).

Software that manages the cache directly is often exposed to the cache block size. Ideally, software should treat the cache block size as an implementation-specific parameter and provide mechanisms to allow it to be set appropriately for the target implementation.

However, it is recognized that there can be circumstances where this approach is not sufficiently practical or efficient. Where binary-level software compatibility for cache instructions is required across a set of implementations of the architecture, the cache block size of those implementations will be the same. Conversely, should the cache block size of two implementations differ, there is no guarantee of binary-level software compatibility for cache instructions between those implementations.

An instruction cache contains instruction cache blocks. An operand cache contains operand cache blocks. In a unified cache, each cache block is both an instruction and operand cache block, and can be used for both instruction and data accesses.

18.5 Cache sets, ways and associativity

A cache block is replicated to form a set. The value used to select a cache block from a set is called the way. The number of ways is given by the set size and is denoted nways, where nways is a power-of-2 and greater than 0.

A set is then replicated to form a cache. The value used to select a set from a cache is called the index. The number of sets is denoted nsets, where nsets is a power-of-2 and greater than 0.

The process of mapping a memory block into the cache is described in *Section 18.6 Cache mapping on page 301*. Essentially, the address of a memory block determines which set the memory block can be cached into.

The memory block can be cached into any of the cache blocks within the selected set. The number of different cache blocks into which a particular memory block can be mapped is known as the associativity of the cache, and is given by nways.

Associativity is a key cache parameter. Increasing the associativity allows more flexibility in the mapping of memory blocks to cache blocks and reduces cache hot-spots. However, increasing the associativity leads to more costly cache implementations and potentially slower cache access times.

The following arrangements are possible:

1 If nways=1, then this is a direct-mapped cache. A memory block can be mapped into exactly one cache block in the cache.

2 If nways>1 and nsets>1, then this is a set-associative cache. A memory block can be mapped into any of the nways cache blocks in a particular set in the cache.

3 If nways>1 and nsets=1, then this is a fully-associative cache. A memory block can be mapped into any of the cache blocks in the cache.

Note that each of these arrangements corresponds to a particular selection of the nways and nsets parameters. This parameterization covers all 3 arrangements.

The cache size in bytes is given by multiplying the cache block size by the set size by the number of sets. If an implementation provides separate instruction and operand caches, then the set size and number of sets can differ for each cache.

18.6 Cache mapping

The mapping of memory blocks to cache blocks is based on the address of the memory block. An address is split into an offset, an index and a tag as shown in *Figure 69*.

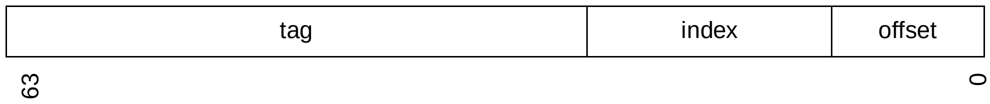


Figure 69: Address decomposition

The boundaries between these fields are determined by the implementation-specific properties already described. The fields are used as follows:

1 The offset selects a byte within the cache block. The number of bits in the offset field is $\log_2$(nbytes).

2 The index selects a set within the cache. The number of bits in the index field is $\log_2$(nsets).

3 The tag consists of all of the remaining address bits. The number of bits in the tag field is 64 - $\log_2$(nsets) - $\log_2$(nbytes).

The mapping of an address proceeds by subscripting into the cache by the index to identify a set. This set consists of all of the cache blocks that this address can be mapped to.

The implementation determines whether this mapping is based on the effective address or the physical address of the access. If there is no translation or if there is an identity translation, then this distinction is immaterial. Additionally, if the smallest page size of the implementation is such that the index of the address is unchanged by the translation process, then again the distinction is mute.

However, if these properties are not all upheld, then the behavior of these approaches is different. The implementation must state whether it indexes after translation (using the physical address), or indexes before translation (using the effective address).

If indexing is achieved using the effective address then cache synonyms must be considered (see *Section 17.8.4 Cache synonyms on page 292*). Cache synonyms occur when the use of an effective address index causes a physical location to be mapped into multiple different sets in the cache. The architecture allows instruction cache synonyms to occur since the instruction cache does not support writes. However, the operand cache supports writes and the presence of operand cache synonyms can lead to incorrect cache operation.

The MMU architecture solves this potential problem by placing constraints on data translations (see *Section 17.8.7 Constraints to avoid cache synonyms on page 294*). If these constraints are honored then all effective address space translations of a particular address will index into the same set and operand cache synonyms are avoided. If these constraints are not honored then accesses through different translations in the effective address space of a particular physical address can be mapped into different sets leading to multiple copies of some memory locations in the operand cache. The operand cache provides no mechanisms to keep these cache synonyms coherent, and this will lead to an unpredictable and faulty memory model.

When an address is held in a particular cache block in a set, tag information is recorded in the cache block to identify this particular address. The index and offset fields need not be recorded as their value is inherent in the cache structure.

The tag information that is associated with each cache block is implementation dependent, and can be different for the instruction cache and operand cache. The available choices are:

- Each cache block is associated with a tag from the physical address of the access.
- Each cache block is associated with a tag from the effective address of the access.
- Each cache block is associated with a tag from the effective address of the access and with a tag from the physical address of the access.

Implementations that use a tag from the effective address behave as described in *Section 18.10 Cache aliases on page 309* with regard to cache aliases.

If the implementation does not implement all of the address space, then some of the upper tag bits will be redundant. For tags derived from the effective address the top (64-neff) bits of the tag are redundant where neff bits of effective address space are implemented. Similarly, for tags derived from the physical address the top (64-nphys) bits of the tag are redundant where nphys bits of physical address space are implemented.

18.7 Caches and memory management

There are significant interactions between the cache and MMU architectures. These are described in *Section 17.8 MMU and caches on page 289*. The main points are:

- Normal cache operation is only provided when the MMU is enabled.
- Constraints are placed on MMU configuration for data translation to avoid the cache synonym problems described in *Section 17.8.7 Constraints to avoid cache synonyms on page 294* and *Section 18.6 Cache mapping on page 301*.
- Changing page table entries typically requires software management of the cache.

18.8 Cache operation

This section describes the general operation of the cache. Subsequent sections will enhance this description by specifying additional cache mechanisms.

18.8.1 Initial state

After a power-on reset, the values of all cache state and all cache configuration registers are architecturally undefined. However, the MMU is disabled and this ensures that all cache state is bypassed and frozen with respect to instruction fetches, data accesses and swap accesses. The semantics of cache operation with the MMU disabled are described in *Section 17.8.1 Cache behavior when the MMU is disabled on page 289*.

The cache should be configured appropriately before the MMU is enabled. This requires that the caches are invalidated and appropriate values are given to the cache configuration registers.

Once the MMU is enabled, the cache becomes enabled. The cache behavior of accesses is then determined by the MMU and cache configurations.

There are various circumstances under which the MMU can be disabled. This can be due to an RTE instruction, a CPU reset, a panic, a debug exception or a debug interrupt. When the MMU is disabled, the cache returns to its frozen and bypassed state regardless of the cache configuration.

18.8.2 Cache access

All read and write accesses supported by the architecture act on up to 8 bytes of data held in an 8-byte-aligned grain of memory. Since the cache block size is at least 8 bytes, then each access falls within a single cache block on all implementations of the architecture.

Operand caches support write-through and write-back behaviors:

- For write-through, each write access updates any associated cache block and is then also propagated through to memory. A property of this approach is that write-through cache blocks are always a copy of the memory state, and can be discarded without requiring any further memory update.
- For write-back, write accesses can be performed on the cache block and the write to memory is postponed until that cache block is discarded. Write-back cache behavior uses a bit in each cache block to distinguish clean and dirty data. Write-back allows aggregation of write accesses to a particular cache block.

The generic behavior of the cache for cachable read and write accesses is as follows:

1 The address of the access is mapped to a set in the cache through the indexing procedure described in *Section 18.6 Cache mapping on page 301*.

2 Each cache block in the set is checked to see if its tag matches the tag of the access. The cache look-up and replacement algorithm is designed so that there can be at most one match in the set.

3 There are two possible outcomes of the tag match:

3.1 If there is no match then this is a cache miss. An implementation-defined replacement algorithm (see *Section 18.8.4 Cache replacement on page 308*) is used to select an appropriate cache block in the set. If there is no replaceable cache block, then the access is performed on memory and there is no change to the cache state. If there is a replaceable cache block, then that cache block is replaced. If that cache block is clean, then it can simply be reused; however, if that cache block is dirty, then its data must be written back out to memory before it is reused. The cache block is marked as clean and refilled from the memory address of this access, then the access continues as if the tag had matched.

3.2 If there is a match, then this is a cache hit. Read accesses simply return the appropriate bytes from the cache block. Write accesses update the appropriate bytes in the cache block. For write-through behavior, a write updates both the cache block state and the memory state. For write-back behavior, a write updates just the cache block state and marks the cache block as dirty.

The behavior for other accesses can differ from the above:

- Swap accesses are described in *Section 6.5.1 Atomic swap on page 98*.
- Prefetch accesses are described in *Section 6.6.1 Prefetch on page 102*.
- Allocate accesses are described in *Section 6.6.2 Allocate on page 103*.
- Cache coherency instructions are described in *Section 6.6.3 Cache coherency on page 104*.

18.8.3 Cache behavior

The cache behavior of an instruction fetch, or data access is determined as follows:

- If the MMU is disabled, then the access bypasses the cache.
- If the MMU is enabled, then the cache behavior is determined by the global cache behavior (specified in cache configuration registers) and by the page-level cache behavior (specified in the PTE for that access). These two behaviors are combined by choosing the more restrictive behavior to give the resultant cache behavior. The combination is specified in *Table 170* and *Table 171*.

Global cache behavior (as specified by cache configuration registers)	Page-level cache behavior (see *Table 166: Cache behavior field on page 283*)	
	Uncachable page	Cachable page
Instruction caching disabled	UNCACHED	UNCACHED
Instruction caching enabled	UNCACHED	CACHED

Table 170: Instruction cache behavior resolution

Global cache behavior (as specified by cache configuration registers)	Page-level cache behavior (see *Table 166: Cache behavior field on page 283*)			
	Device page	Uncachable page	Cachable, write-through page	Cachable, write-back page
Data caching disabled	DEVICE	UNCACHED	UNCACHED	UNCACHED
Data caching enabled, data write-back disabled	DEVICE	UNCACHED	WRITE-THROUGH	WRITE-THROUGH
Data caching enabled, data write-back enabled	DEVICE	UNCACHED	WRITE-THROUGH	WRITE-BACK

Table 171: Operand cache behavior resolution

Cache behavior is a property of a physical page in memory. Software must ensure that all accesses to a particular physical page use compatible cache behaviors. All data accesses to a physical page must use the same operand cache behavior. All instruction fetches from a physical page must use the same instruction cache behavior.

Cache behavior can be selected independently for instruction accesses and data accesses, but there are restrictions on the allowed combinations for a particular physical page. If a physical page is uncachable for instructions, then it must have either device or uncached behavior for data. If a physical page is cachable for instructions, then it must have either write-through or write-back behavior for data. These restrictions are necessary to ensure correct behavior on implementations with a unified cache.

The properties of the resultant cache behaviors are described in the following sections.

Uncached instruction

Accesses with this behavior are performed directly on the memory system. Uncached instructions are never placed in the cache, and therefore these accesses never hit the cache nor change the state of the cache. An implementation can optimize these accesses. The implementation can transfer more data than that specified in the access, and can aggregate the access with other accesses.

Cached instruction

Accesses with this behavior are performed through the cache. These accesses can hit the cache and can allocate clean cache blocks. An implementation can optimize these accesses. The implementation can transfer more data than that specified in the access, and can aggregate the access with other accesses.

Device data

Accesses with this behavior are performed directly on the memory system. Device data is never placed in the cache, and therefore these accesses never hit the cache nor change the state of the cache. An implementation does not optimize device accesses. The precise amount of data specified in the access is transferred and the access is not aggregated with any other. Note that it is necessary to use the SYNCO instruction (see *Section 6.5.3 Data synchronization on page 99*) to impose ordering on device accesses where required.

Uncached data

Accesses with this behavior are performed directly on the memory system. Uncached data is never placed in the cache, and therefore these accesses never hit the cache nor change the state of the cache. An implementation can optimize these accesses. The implementation can transfer more data than that specified in the access, and can aggregate the access with other accesses.

Write-through data

Accesses with this behavior are performed through the cache using write-through semantics. These accesses can hit the cache and can allocate clean cache blocks. Dirty data is never placed in the cache, and therefore these accesses never hit on dirty data. An implementation can optimize these accesses. The implementation can transfer more data than that specified in the access, and can aggregate the access with other accesses.

Write-back data

Accesses with this behavior are performed through the cache using write-back semantics. These accesses can hit the cache and can allocate clean or dirty cache blocks. An implementation can optimize these accesses. The implementation can transfer more data than that specified in the access, and can aggregate the access with other accesses.

18.8.4 Cache replacement

When a cachable access misses the cache, the cache replacement algorithm is used to determine which, if any, cache block is to be evicted from the cache to allow the new access to be cached. The address of the access is used to index into the cache (as described in *Section 18.6 Cache mapping on page 301*) and select a set. There will be nways cache blocks in the selected set, and these are candidates for replacement.

The details of the cache replacement algorithm are implementation specific. Typical algorithms maintain some additional state for each set to allow the choice to be influenced by the recent access history to that set. A common algorithm is to select the cache block which has been least-recently-used.

18.8.5 Cache locking

Optionally, an implementation can choose to provide a cache locking feature. This is a mechanism that allows data to be loaded into cache blocks and then locked. Locked cache blocks are not eligible for replacement and will therefore remain in the cache until explicitly discarded.

If it is possible to lock all cache blocks in a particular set, then the replacement algorithm will find no replaceable blocks. Any cache miss for that set will be performed on memory without caching.

All other details of cache locking are implementation specific.

18.9 Cache paradoxes

When the MMU is enabled, inappropriate use of cache behavior can result in an access finding the cache in an inconsistent state. These states are called cache paradoxes.

Cache behavior is determined by page-level cache behavior and global cache behavior as described in *Section 18.8.3 Cache behavior on page 305*. Inappropriate management of page-level or global cache behavior can lead to cache paradoxes.

The following situations must be avoided:

- An instruction access using ‘UNCACHED INSTRUCTION’ behavior hits the cache.

- A data access using ‘DEVICE’ behavior hits the cache.
- A data access using ‘UNCACHED DATA’ behavior hits the cache.
- A data access using ‘WRITE-THROUGH DATA’ behavior hits the cache and the cache block is dirty.

The behavior of these accesses is architecturally undefined. Software must explicitly cohere the cache to avoid these situations when the cache behavior of a particular physical page is changed.

When the MMU is disabled, the state of the cache is bypassed and frozen, and cache paradoxes cannot occur. A possible scenario is for software to be running with the MMU enabled, to then disable the MMU for some reason, and to subsequently re-enable the MMU. If software requires a coherent memory model through this sequence, then coherency must be achieved in software through appropriate cache management.

18.10 Cache aliases

The architecture allows cache blocks to be tagged by either physical addresses or by effective addresses (see *Section 18.6 Cache mapping on page 301*).

When a cache implementation uses effective addresses to tag cache blocks, the potential problem of cache aliases must be considered. The MMU architecture allows a particular physical address to be mapped into multiple effective addresses and in multiple effective address spaces. The issue is whether these address space aliases can result in multiple cache blocks to be simultaneously valid in the same set for a particular physical address; that is, whether the cache can contain cache aliases. If cache aliases are allowed, then coherency of those aliases has to be considered.

The architecture takes the following position on cache aliases:

- Cache aliases are guaranteed not to exist for operand cache blocks. The implementation provides transparent mechanisms to resolve operand cache aliases, such that there is guaranteed to be at most one operand cache block in a particular set corresponding to any physical address.

  The mechanisms used to achieve this are implementation-specific. One implementation choice is to use a direct-mapped operand cache since this contains only one cache block per set. Another implementation choice is to record both the effective address tag and physical address tag in each cache block, and use the physical address tags to detect and avoid cache aliases. Both of these arrangements ensure that there can be at most one operand cache block in a particular set corresponding to any physical address. The architecture does not limit implementations to these approaches.

- Cache aliases can exist for instruction cache blocks. An implementation is not required to provide mechanisms to resolve instruction cache aliases. There can be multiple instruction cache blocks in a particular set that correspond to the same physical address.

There are asymmetries between the policies for cache alias resolution for the operand cache and the instruction cache. This is because the instruction cache does not support writes, and multiple copies of instructions do not lead to incoherency in the instruction cache. However, this property is visible to software through the ICBI instruction (see *Section 6.6.3 Cache coherency on page 104*).

The architecture recognizes a related, but distinct, property of caches. This is the concept of cache synonyms, which is described separately in *Section 17.8.4 Cache synonyms on page 292*. Cache synonyms are caused by indexing into the cache using an index from the effective address (see *Section 18.6*) and are solved by software constraints not by hardware mechanism.

Section 17.8 MMU and caches on page 289 describes the constraints which must be obeyed to avoid cache synonyms in the operand cache. If the MMU and cache architectures are used correctly, then neither cache synonyms nor cache aliases will exist in the operand cache. This ensures correct cache operation for data. The MMU and cache architectures allow both cache synonyms and cache aliases to exist for the instruction cache, since they do not lead to incorrect instruction cache operation. However, software must consider instruction cache synonyms and aliases when invalidating instructions.

The architecture chooses to use different names for ‘cache synonyms’ and ‘cache aliases’ because very different techniques are used in the architecture to solve these distinct problems. Cache synonyms are multiple copies of the same physical location occurring in different sets due to indexing into the cache using the effective address. Cache aliases are multiple copies of the same physical location occurring in the same set due to the use of effective address tags in the cache. The use of different names for these concepts leads to a clearer and simpler description. Please note that some other architectures use these terms interchangeably without making any such distinction.

18.11 Speculative memory accesses

A speculative memory access is an access made by the implementation that is not required by the abstract sequential model of instruction execution. This model is described in *Volume 2 Chapter 1: SHmedia specification*. Memory accesses that result from data or instruction prefetch are also considered speculative.

The architecture places limits on speculative memory accesses. This allows software to be arranged so that device memory is not exposed to speculative memory access.

The architectural limits depend on whether the MMU is enabled or disabled. The limits are defined in terms of accesses to external memory (that is, beyond the cache).

18.11.1 Speculative memory access when MMU is enabled

When the MMU is enabled, speculative memory access is controlled by protection attributes (see *Protection (PTEL.PR) on page 284*) and resultant cache behavior (see *Section 18.8.3 Cache behavior on page 305*).

In general, an implementation is only allowed to perform speculative memory accesses that are consistent with the protection and cache behavior settings for a page.

The architecture allows an implementation to perform:

- Speculative instruction fetch from a page with an active executable translation.
- Speculative data loads from a page with an active accessible translation which has a cache behavior of UNCACHED, WRITE-THROUGH or WRITE-BACK DATA.
- Speculative data stores to a page with an active writable translation which has a cache behavior of UNCACHED, WRITE-THROUGH or WRITE-BACK DATA.

The architecture disallows an implementation from performing:

- Speculative instruction fetch from a page without an active executable instruction translation.
- Speculative data loads from a page without an active accessible data translation.
- Speculative data stores to a page without an active writable data translation.
- Speculative data loads or stores from a page with an active data translation which has a cache behavior of DEVICE.

In the above description, an active translation corresponds to a PTE that the MMU is aware of (see *Section 17.8.3 Cache coherency when changing the page table on page 291*).

Executable means that the translation has executability for the current privilege. *Accessible* means that the translation has readability or writability for the current privilege. *Writable* means that the translation has writability for the current privilege.

The architecture is arranged so that data and instruction prefetches are guaranteed to have no effect in the cases above where speculative access is not allowed.

18.11.2 Speculative memory access when MMU is disabled

When the MMU is disabled, speculative access cannot be controlled by page table entries. Speculative accesses are limited as follows:

- All data accesses, including swap accesses, are implemented as though they were device accesses. The data cache is frozen and bypassed. The precise amount of data is transferred. There is no speculative data access.
- Instruction fetches are not cached. The instruction cache is frozen and bypassed. Additionally, the amount of speculative instruction fetch is restricted to avoid prefetches from device areas of physical memory.

The architecture allows an implementation to perform speculative fetch of instructions that are in:

- the page that encloses the program counter.
- the page immediately following the page that encloses the program counter.
- the pages that enclose each of the locations referred to by the 8 target registers.

In each case above, the page has the smallest page size that is supported by the implementation and starts at an effective address that is aligned to that page size.

When the MMU is disabled, software must prevent target registers from referring to device memory. However, note that after POWERON reset, target registers contain undefined values and the MMU is disabled. All implementations guarantee not to prefetch from these undefined addresses as long as a branch instruction is not executed. Therefore, after a POWERON reset software must initialize the target registers before branch instructions are executed.

The following sequence is recommended:

```
PTABS/U R63, TR0
PTABS/U R63, TR1
PTABS/U R63, TR2
PTABS/U R63, TR3
PTABS/U R63, TR4
PTABS/U R63, TR5
PTABS/U R63, TR6
PTABS/U R63, TR7
SYNCI
```

The effect of the 8 PTABS instructions is to give a defined value to each target register. The SYNCI instruction prevents speculative execution of subsequent instructions (see *Section 6.5.2 Instruction synchronization on page 99*). It ensures that all target registers are properly defined before any subsequent branch instruction can cause instruction prefetch.

This initialization should occur very early in the software boot-strap.

SuperH

SHmedia summary

A

Instruction	Summary
ADD Rm,Rn,Rd	add 64-bit
ADD.L Rm,Rn,Rd	add 32-bit
ADDI Rm,imm,Rd	add immediate 64-bit
ADDI.L Rm,imm,Rd	add immediate 32-bit
ADDZ.L Rm,Rn,Rd	add with zero-extend 32-bit
ALLOCO Rm,disp	allocate operand cache block
AND Rm,Rn,Rd	bitwise AND 64-bit
ANDC Rm,Rn,Rd	bitwise ANDC 64-bit
ANDI Rm,imm,Rd	bitwise AND immediate 64-bit
BEQ Rm,Rn,TRc	branch if equal 64-bit
BEQI Rm,imm,TRc	branch if equal to immediate 64-bit
BGE Rm,Rn,TRc	branch if greater than or equal 64-bit signed
BGEU Rm,Rn,TRc	branch if greater than or equal 64-bit unsigned
BGT Rm,Rn,TRc	branch if greater than 64-bit signed
BGTU Rm,Rn,TRc	branch if greater than 64-bit unsigned

Table 172: SHmedia instruction set summary

Instruction	Summary
BLINK TRb,Rd	branch unconditionally and link
BNE Rm,Rn,TRc	branch if not equal 64-bit
BNEI Rm,imm,TRc	branch if not equal to immediate 64-bit
BRK	cause a break
BYTEREV Rm,Rd	byte reversal
CMPEQ Rm,Rn,Rd	compare equal 64-bit
CMPGT Rm,Rn,Rd	compare greater than 64-bit signed
CMPGTU Rm,Rn,Rd	compare greater than 64-bit unsigned
CMVEQ Rm,Rn,Rw	conditional move if equal to zero
CMVNE Rm,Rn,Rw	conditional move if not equal to zero
FABS.D DRg,DRf	get absolute value of a double-precision number
FABS.S FRg,FRf	get absolute value of a single-precision number
FADD.D DRg,DRh,DRf	add two double-precision numbers
FADD.S FRg,FRh,FRf	add two single-precision numbers
FCMPEQ.D DRg,DRh,Rd	compare double-precision numbers for equality
FCMPEQ.S FRg,FRh,Rd	compare single-precision numbers for equality
FCMPGE.D DRg,DRh,Rd	compare double-precision numbers for greater-or-equal
FCMPGE.S FRg,FRh,Rd	compare single-precision numbers for greater-or-equal
FCMPGT.D DRg,DRh,Rd	compare double-precision numbers for greater-than
FCMPGT.S FRg,FRh,Rd	compare single-precision numbers for greater-than
FCMPUN.D DRg,DRh,Rd	compare double-precision numbers for unorderedness
FCMPUN.S FRg,FRh,Rd	compare single-precision numbers for unorderedness
FCNV.DS DRg,FRf	double-precision to single-precision conversion
FCNV.SD FRg,DRf	single-precision to double-precision conversion
FCOSA.S FRg,FRf	approximate cosine of an angle

Table 172: SHmedia instruction set summary

Instruction	Summary
FDIV.D DRg,DRh,DRf	divide two double-precision numbers
FDIV.S FRg,FRh,FRf	divide two single-precision numbers
FGETSCR FRf	move from floating-point status/control register
FIPR.S FVg,FVh,FRf	compute inner (dot) product of two vectors
FLD.D Rm,disp,DRf	load 64-bit value
FLD.P Rm,disp,FPf	load two 32-bit values
FLD.S Rm,disp,FRf	load 32-bit value
FLDX.D Rm,Rn,DRf	load indexed 64-bit value
FLDX.P Rm,Rn,FPf	load indexed two 32-bit values
FLDX.S Rm,Rn,FRf	load indexed 32-bit value
FLOAT.LD FRg,DRf	32-bit integer to double-precision conversion
FLOAT.LS FRg,FRf	32-bit integer to single-precision conversion
FLOAT.QD DRg,DRf	64-bit integer to double-precision conversion
FLOAT.QS DRg,FRf	64-bit integer to single-precision conversion
FMAC.S FRg,FRh,FRq	single-precision fused multiply accumulate
FMOV.D DRg,DRf	64-bit floating-point to floating-point register move
FMOV.DQ DRg,Rd	64-bit floating-point to general register move
FMOV.LS Rm,FRf	32-bit general to floating-point register move
FMOV.QD Rm,DRf	64-bit general to floating-point register move
FMOV.S FRg,FRf	32-bit floating-point to floating-point register move
FMOV.SL FRg,Rd	32-bit floating-point to general register move
FMUL.D DRg,DRh,DRf	multiply two double-precision numbers
FMUL.S FRg,FRh,FRf	multiply two single-precision numbers
FNEG.D DRg,DRf	negate a double-precision number
FNEG.S FRg,FRf	negate a single-precision number

Table 172: SHmedia instruction set summary

Instruction	Summary
FPUTSCR FRg	move to floating-point status/control register
FSINA.S FRg,FRf	approximate sine of an angle
FSQRT.D DRg,DRf	find square root of a double-precision number
FSQRT.S FRg,FRf	find square root of a single-precision number
FSRRA.S FRg,FRf	approximate reciprocal of a square root of a value
FST.D Rm,disp,DRz	store 64-bit value
FST.P Rm,disp,FPz	store two 32-bit values
FST.S Rm,disp,FRz	store 32-bit value
FSTX.D Rm,Rn,DRz	store indexed 64-bit value
FSTX.P Rm,Rn,FPz	store indexed two 32-bit values
FSTX.S Rm,Rn,FRz	store indexed 32-bit value
FSUB.D DRg,DRh,DRf	subtract two double-precision numbers
FSUB.S FRg,FRh,FRf	subtract two single-precision numbers
FTRC.DL DRg,FRf	double-precision to 32-bit integer conversion
FTRC.SL FRg,FRf	single-precision to 32-bit integer conversion
FTRC.DQ DRg,DRf	double-precision to 64-bit integer conversion
FTRC.SQ FRg,DRf	single-precision to 64-bit integer conversion
FTRV.S MTRXg,FVh,FVf	transform vector
GETCFG Rm,disp,Rd	move from configuration register
GETCON CRk,Rd	move from control register
GETTR TRb,Rd	move from target register
ICBI Rm,disp	instruction cache block invalidate
LD.B Rm,disp,Rd	load 8-bit signed
LD.L Rm,disp,Rd	load 32-bit
LD.Q Rm,disp,Rd	load 64-bit

Table 172: SHmedia instruction set summary

Instruction	Summary
LD.UB Rm,disp,Rd	load 8-bit unsigned
LD.UW Rm,disp,Rd	load 16-bit unsigned
LD.W Rm,disp,Rd	load 16-bit signed
LDHI.L Rm,disp,Rd	load misaligned high part 32-bit
LDHI.Q Rm,disp,Rd	load misaligned high part 64-bit
LDLO.L Rm,disp,Rd	load misaligned low part 32-bit
LDLO.Q Rm,disp,Rd	load misaligned low part 64-bit
LDX.B Rm,Rn,Rd	load indexed 8-bit signed
LDX.L Rm,Rn,Rd	load indexed 32-bit
LDX.Q Rm,Rn,Rd	load indexed 64-bit
LDX.UB Rm,Rn,Rd	load indexed 8-bit unsigned
LDX.UW Rm,Rn,Rd	load indexed 16-bit unsigned
LDX.W Rm,Rn,Rd	load indexed 16-bit signed
MABS.L Rm,Rd	multimedia absolute value signed 32-bit with saturation
MABS.W Rm,Rd	multimedia absolute value signed 16-bit with saturation
MADD.L Rm,Rn,Rd	multimedia add 32-bit
MADD.W Rm,Rn,Rd	multimedia add 16-bit
MADDS.L Rm,Rn,Rd	multimedia add signed 32-bit with saturation
MADDS.UB Rm,Rn,Rd	multimedia add unsigned 8-bit with saturation
MADDS.W Rm,Rn,Rd	multimedia add signed 16-bit with saturation
MCMPEQ.B Rm,Rn,Rd	multimedia compare equal 8-bit
MCMPEQ.L Rm,Rn,Rd	multimedia compare equal 32-bit
MCMPEQ.W Rm,Rn,Rd	multimedia compare equal 16-bit
MCMPGT.L Rm,Rn,Rd	multimedia compare greater than signed 32-bit
MCMPGT.UB Rm,Rn,Rd	multimedia compare greater than unsigned 8-bit

Table 172: SHmedia instruction set summary

Instruction	Summary
MCMPGT.W Rm,Rn,Rd	multimedia compare greater than signed 16-bit
MCMV Rm,Rn,Rw	multimedia bitwise conditional move
MCNVS.LW Rm,Rn,Rd	multimedia convert signed 32-bit to signed 16-bit after saturation
MCNVS.WB Rm,Rn,Rd	multimedia convert signed 16-bit to signed 8-bit after saturation
MCNVS.WUB Rm,Rn,Rd	multimedia convert signed 16-bit to unsigned 8-bit after saturation
MEXTR1 Rm,Rn,Rd	multimedia extract 64 bits from 128 bits using a 1x8-bit offset
MEXTR2 Rm,Rn,Rd	multimedia extract 64 bits from 128 bits using a 2x8-bit offset
MEXTR3 Rm,Rn,Rd	multimedia extract 64 bits from 128 bits using a 3x8-bit offset
MEXTR4 Rm,Rn,Rd	multimedia extract 64 bits from 128 bits using a 4x8-bit offset
MEXTR5 Rm,Rn,Rd	multimedia extract 64 bits from 128 bits using a 5x8-bit offset
MEXTR6 Rm,Rn,Rd	multimedia extract 64 bits from 128 bits using a 6x8-bit offset
MEXTR7 Rm,Rn,Rd	multimedia extract 64 bits from 128 bits using a 7x8-bit offset
MMACFX.WL Rm,Rn,Rw	multimedia fractional multiply and accumulate signed 16-bit with saturation
MMACNFX.WL Rm,Rn,Rw	multimedia fractional multiply and subtract signed 16-bit with saturation
MMUL.L Rm,Rn,Rd	multimedia multiply 32-bit
MMUL.W Rm,Rn,Rd	multimedia multiply 16-bit
MMULFX.L Rm,Rn,Rd	multimedia fractional multiply signed 32-bit
MMULFX.W Rm,Rn,Rd	multimedia fractional multiply signed 16-bit
MMULFXRP.W Rm,Rn,Rd	multimedia fractional multiply signed 16-bit, round nearest positive
MMULHI.WL Rm,Rn,Rd	multimedia full multiply signed 16-bit high
MMULLO.WL Rm,Rn,Rd	multimedia full multiply signed 16-bit low
MMULSUM.WQ Rm,Rn,Rw	multimedia multiply and sum signed 16-bit
MOVI imm,Rd	move immediate
MPERM.W Rm,Rn,Rd	multimedia permute 16-bits
MSAD.UBQ Rm,Rn,Rw	multimedia sum of absolute differences of unsigned 8-bit

Table 172: SHmedia instruction set summary

Instruction	Summary
MSHALDS.L Rm,Rn,Rd	multimedia shift arithmetic left dynamic 32-bit with saturation
MSHALDS.W Rm,Rn,Rd	multimedia shift arithmetic left dynamic 16-bit with saturation
MSHARD.L Rm,Rn,Rd	multimedia shift arithmetic right dynamic 32-bit
MSHARD.W Rm,Rn,Rd	multimedia shift arithmetic right dynamic 16-bit
MSHARDS.Q Rm,Rn,Rd	multimedia shift arithmetic right dynamic with saturation to signed 16-bit
MSHFHI.B Rm,Rn,Rd	multimedia shuffle upper-half 8-bit
MSHFHI.L Rm,Rn,Rd	multimedia shuffle upper-half 32-bit
MSHFHI.W Rm,Rn,Rd	multimedia shuffle upper-half 16-bit
MSHFLO.B Rm,Rn,Rd	multimedia shuffle lower-half 8-bit
MSHFLO.L Rm,Rn,Rd	multimedia shuffle lower-half 32-bit
MSHFLO.W Rm,Rn,Rd	multimedia shuffle lower-half 16-bit
MSHLLD.L Rm,Rn,Rd	multimedia shift logical left dynamic 32-bit
MSHLLD.W Rm,Rn,Rd	multimedia shift logical left dynamic 16-bit
MSHLRD.L Rm,Rn,Rd	multimedia shift logical right dynamic 32-bit
MSHLRD.W Rm,Rn,Rd	multimedia shift logical right dynamic 16-bit
MSUB.L Rm,Rn,Rd	multimedia subtract 32-bit
MSUB.W Rm,Rn,Rd	multimedia subtract 16-bit
MSUBS.L Rm,Rn,Rd	multimedia subtract signed 32-bit with saturation
MSUBS.UB Rm,Rn,Rd	multimedia subtract unsigned 8-bit with saturation
MSUBS.W Rm,Rn,Rd	multimedia subtract signed 16-bit with saturation
MULS.L Rm,Rn,Rd	multiply full 32-bit x 32-bit to 64-bit signed
MULU.L Rm,Rn,Rd	multiply full 32-bit x 32-bit to 64-bit unsigned
NOP	no operation
NSB Rm,Rd	count number of sign bits
OCBI Rm,disp	operand cache block invalidate

Table 172: SHmedia instruction set summary

Instruction	Summary
OCBP Rm,disp	operand cache block purge
OCBWB Rm,disp	operand cache block write-back
OR Rm,Rn,Rd	bitwise OR 64-bit
ORI Rm,imm,Rd	bitwise OR immediate 64-bit
PREFI Rm,disp	prefetch instruction cache block
PTA label,TRa	prepare target relative immediate (target is SHmedia)
PTABS Rn,TRa	prepare target absolute register
PTB label,TRa	prepare target relative immediate (target is SHcompact)
PTREL Rn,TRa	prepare target relative register
PUTCFG Rm,disp,Ry	move to configuration register
PUTCON Rm,CRj	move to control register
RTE	return from exception
SHARD Rm,Rn,Rd	shift arithmetic right dynamic 64-bit
SHARD.L Rm,Rn,Rd	shift arithmetic right dynamic 32-bit
SHARI Rm,imm,Rd	shift arithmetic right immediate 64-bit
SHARI.L Rm,imm,Rd	shift arithmetic right immediate 32-bit
SHLLD Rm,Rn,Rd	shift logical left dynamic 64-bit
SHLLD.L Rm,Rn,Rd	shift logical left dynamic 32-bit
SHLLI Rm,imm,Rd	shift logical left immediate 64-bit
SHLLI.L Rm,imm,Rd	shift logical left immediate 32-bit
SHLRD Rm,Rn,Rd	shift logical right dynamic 64-bit
SHLRD.L Rm,Rn,Rd	shift logical right dynamic 32-bit
SHLRI Rm,imm,Rd	shift logical right immediate 64-bit
SHLRI.L Rm,imm,Rd	shift logical right immediate 32-bit
SHORI imm,Rw	shift then or immediate

Table 172: SHmedia instruction set summary

Instruction	Summary
SLEEP	enter sleep mode
ST.B Rm,disp,Ry	store 8-bit
ST.L Rm,disp,Ry	store 32-bit
ST.Q Rm,disp,Ry	store 64-bit
ST.W Rm,disp,Ry	store 16-bit
STHI.L Rm,disp,Ry	store misaligned high part 32-bit
STHI.Q Rm,disp,Ry	store misaligned high part 64-bit
STLO.L Rm,disp,Ry	store misaligned low part 32-bit
STLO.Q Rm,disp,Ry	store misaligned low part 64-bit
STX.B Rm,Rn,Ry	store indexed 8-bit
STX.L Rm,Rn,Ry	store indexed 32-bit
STX.Q Rm,Rn,Ry	store indexed 64-bit
STX.W Rm,Rn,Ry	store indexed 16-bit
SUB Rm,Rn,Rd	subtract 64-bit
SUB.L Rm,Rn,Rd	subtract 32-bit
SWAP.Q Rm,Rn,Rw	atomic swap in memory 64-bit
SYNCI	synchronize instructions
SYNCO	synchronize operand data
TRAPA Rm	cause a trap
XOR Rm,Rn,Rd	bitwise XOR 64-bit
XORI Rm,imm,Rd	bitwise XOR immediate 64-bit

Table 172: SHmedia instruction set summary

SuperH

SHcompact summary

B

Instruction	Summary
ADD Rm, Rn	add
ADD #imm, Rn	add immediate
ADDC Rm, Rn	add with carry
ADDV Rm, Rn	add with overflow check
AND Rm, Rn	bitwise AND
AND #imm, R0	bitwise AND immediate
AND.B #imm, @(R0, GBR)	bitwise AND memory
BF label	branch if T-bit is false
BF/S label	delayed branch if T-bit is false
BRA label	delayed branch
BRAF Rn	delayed branch far
BRK	cause a break
BSR label	delayed branch to subroutine
BSRF Rn	delayed branch to subroutine far
BT label	branch if T-bit is true

Table 173: SHcompact instruction set summary

Instruction	Summary
BT/S label	delayed branch if T-bit is true
CLRMAC	clear MACL and MACH registers
CLRS	clear S-bit
CLRT	clear T-bit
CMP/EQ Rm, Rn	compare equal, result placed in T-bit
CMP/EQ #imm, R0	compare equal immediate, result placed in T-bit
CMP/GE Rm, Rn	compare greater than or equal, result placed in T-bit
CMP/GT Rm, Rn	compare greater than, result placed in T-bit
CMP/HI Rm, Rn	compare higher, result placed in T-bit
CMP/HS Rm, Rn	compare higher same, result placed in T-bit
CMP/PL Rn	compare greater than 0, result placed in T-bit
CMP/PZ Rn	compare greater equal 0, result placed in T-bit
CMP/STR Rm, Rn	compare string, result placed in T-bit
DIV0S Rm, Rn	divide step 0 as signed
DIV0U	divide step 0 as unsigned
DIV1 Rm, Rn	divide step 1
DMULS.L Rm, Rn	double-length multiply as signed
DMULU.L Rm, Rn	double-length multiply as unsigned
DT Rn	decrement and test
EXTS.B Rm, Rn	byte extend as signed
EXTS.W Rm, Rn	word extend as signed
EXTU.B Rm, Rn	byte extend as unsigned
EXTU.W Rm, Rn	word extend as unsigned
FABS DRn	double floating-point absolute value
FABS FRn	single floating-point absolute value

Table 173: SHcompact instruction set summary

Instruction	Summary
FADD DRm, DRn	double floating-point add
FADD FRm, FRn	single floating-point add
FCMP/EQ DRm, DRn	double floating-point compare equal, result placed in T-bit
FCMP/EQ FRm, FRn	single floating-point compare equal, result placed in T-bit
FCMP/GT DRm, DRn	double floating-point compare greater, result placed in T-bit
FCMP/GT FRm, FRn	single floating-point compare greater, result placed in T-bit
FCNVDS DRm, FPUL	double to single floating-point convert
FCNVSD FPUL, DRn	single to double floating-point convert
FDIV DRm, DRn	double floating-point divide
FDIV FRm, FRn	single floating-point divide
FIPR FVm, FVn	single floating-point inner product
FLDI0 FRn	single floating-point load of 0.0
FLDI1 FRn	single floating-point load of 1.0
FLDS FRm, FPUL	floating-point load from register to FPUL
FLOAT FPUL, DRn	double floating-point convert from integer
FLOAT FPUL, FRn	single floating-point convert from integer
FMAC FR0, FRm, FRn	single floating-point multiply and accumulate
FMOV DRm, DRn	single-pair to single-pair floating-point move
FMOV DRm, XDn	single-pair to extended single-pair floating-point move
FMOV DRm, @Rn	single-pair floating-point store indirect
FMOV DRm, @-Rn	single-pair floating-point store indirect with pre-decrement
FMOV DRm, @(R0, Rn)	single-pair floating-point store indirect with indexing
FMOV FRm, FRn	single to single floating-point move
FMOV.S FRm, @Rn	single floating-point store indirect
FMOV.S FRm, @-Rn	single floating-point store indirect with pre-decrement

Table 173: SHcompact instruction set summary

Instruction	Summary
FMOV.S FRm, @(R0, Rn)	single floating-point store indirect with indexing
FMOV XDm, DRn	extended single-pair to single-pair floating-point move
FMOV XDm, XDn	extended single-pair to extended single-pair floating-point move
FMOV XDm, @Rn	extended single-pair floating-point store indirect
FMOV XDm, @-Rn	extended single-pair floating-point store indirect with pre-decrement
FMOV XDm, @(R0, Rn)	extended single-pair floating-point store indirect with indexing
FMOV @Rm, DRn	single-pair floating-point load indirect
FMOV @Rm+, DRn	single-pair floating-point load indirect with post-increment
FMOV @(R0, Rm), DRn	single-pair floating-point load indirect with indexing
FMOV.S @Rm, FRn	single floating-point load indirect
FMOV.S @Rm+, FRn	single floating-point load indirect with post-increment
FMOV.S @(R0, Rm), FRn	single floating-point load indirect with indexing
FMOV @Rm, XDn	extended single-pair floating-point load indirect
FMOV @Rm+, XDn	extended single-pair floating-point load indirect with post-increment
FMOV @(R0, Rm), XDn	extended single-pair floating-point load indirect with indexing
FMUL DRm, DRn	double floating-point multiply
FMUL FRm, FRn	single floating-point multiply
FNEG DRn	double floating-point negate
FNEG FRn	single floating-point negate
FRCHG	FR-bit change (toggle)
FSCA FPUL, DRn	single floating-point sine cosine approximate
FSCHG	SZ-bit change (toggle)
FSQRT DRn	double floating-point square root
FSQRT FRn	single floating-point square root
FSRRA FRn	single reciprocal square root approximate

Table 173: SHcompact instruction set summary

Instruction	Summary
FSTS FPUL, FRn	floating-point store to register from FPUL
FSUB DRm, DRn	double floating-point subtract
FSUB FRm, FRn	single floating-point subtract
FTRC DRm, FPUL	double floating-point truncate and convert to integer
FTRC FRm, FPUL	single floating-point truncate and convert to integer
FTRV XMTRX, FVn	single floating-point transform vector
JMP @Rn	delayed jump
JSR @Rn	delayed jump to subroutine
LDC Rm, GBR	load from register to GBR
LDC.L @Rm+, GBR	load from memory to GBR with post-increment
LDS Rm, FPSCR	load from register to FPSCR
LDS.L @Rm+, FPSCR	load from memory to FPSCR with post-increment
LDS Rm, FPUL	load from register to FPUL
LDS.L @Rm+, FPUL	load from memory to FPUL with post-increment
LDS Rm, MACH	load from register to MACH
LDS.L @Rm+, MACH	load from memory to MACH with post-increment
LDS Rm, MACL	load from register to MACL
LDS.L @Rm+, MACL	load from memory to MACL with post-increment
LDS Rm, PR	load from register to PR
LDS.L @Rm+, PR	load from memory to PR with post-increment
MAC.L @Rm+, @Rn+	multiply and accumulate long
MAC.W @Rm+, @Rn+	multiply and accumulate word
MOV Rm, Rn	move data
MOV #imm, Rn	move immediate data
MOV.B Rm, @Rn	store 8-bits indirect

Table 173: SHcompact instruction set summary

Instruction	Summary
MOV.B Rm, @-Rn	store 8-bits indirect with pre-decrement
MOV.B Rm, @(R0, Rn)	store 8-bits indirect with indexing
MOV.B R0, @(disp, GBR)	store 8-bits indirect to GBR with displacement
MOV.B R0, @(disp, Rn)	store 8-bits indirect with displacement
MOV.B @Rm, Rn	load 8-bits indirect
MOV.B @Rm+, Rn	load 8-bits indirect with post-increment
MOV.B @(R0, Rm), Rn	load 8-bits indirect with indexing
MOV.B @(disp, GBR), R0	load 8-bits indirect from GBR with displacement
MOV.B @(disp, Rm), R0	load 8-bits indirect with displacement
MOV.L Rm, @Rn	store 32-bits indirect
MOV.L Rm, @-Rn	store 32-bits indirect with pre-decrement
MOV.L Rm, @(R0, Rn)	store 32-bits indirect with indexing
MOV.L R0, @(disp, GBR)	store 32-bits indirect to GBR with displacement
MOV.L Rm, @(disp, Rn)	store 32-bits indirect with displacement
MOV.L @Rm, Rn	load 32-bits indirect
MOV.L @Rm+, Rn	load 32-bits indirect with post-increment
MOV.L @(R0, Rm), Rn	load 32-bits indirect with indexing
MOV.L @(disp, GBR), R0	load 32-bits indirect from GBR with displacement
MOV.L @(disp, PC), Rn	load 32-bits indirect from PC with displacement
MOV.L @(disp, Rm), Rn	load 32-bits indirect with displacement
MOV.W Rm, @Rn	store 16-bits indirect
MOV.W Rm, @-Rn	store 16-bits indirect with pre-decrement
MOV.W Rm, @(R0, Rn)	store 16-bits indirect with indexing
MOV.W R0, @(disp, GBR)	store 16-bits indirect to GBR with displacement
MOV.W R0, @(disp, Rn)	store 16-bits indirect with displacement

Table 173: SHcompact instruction set summary

Instruction	Summary
MOV.W @Rm, Rn	load 16-bits indirect
MOV.W @Rm+, Rn	load 16-bits indirect with post-increment
MOV.W @(R0, Rm), Rn	load 16-bits indirect with indexing
MOV.W @(disp, GBR), R0	load 16-bits indirect from GBR with displacement
MOV.W @(disp, PC), Rn	load 16-bits indirect from PC with displacement
MOV.W @(disp, Rm), R0	load 16-bits indirect with displacement
MOVA @(disp, PC), R0	move PC-relative address
MOVCA.L R0, @Rn	store long not fetching block
MOVT Rn	move T-bit
MUL.L Rm, Rn	multiply long
MULS.W Rm, Rn	multiply signed word
MULU.W Rm, Rn	multiply unsigned word
NEG Rm, Rn	negate
NEGC Rm, Rn	negate with carry
NOP	no operation
NOT Rm, Rn	bitwise NOT
OCBI @Rn	operand cache block invalidate
OCBP @Rn	operand cache block purge
OCBWB @Rn	operand cache block writeback
OR Rm, Rn	or logical
OR #imm, R0	bitwise OR immediate
OR.B #imm, @(R0, GBR)	bitwise OR memory
PREF @Rn	prefetch operand data
ROTCL Rn	rotate with carry left
ROTCR Rn	rotate with carry right

Table 173: SHcompact instruction set summary

Instruction	Summary
ROTL Rn	rotate left
ROTR Rn	rotate right
RTS	delayed return from subroutine
SETS	set S-bit
SETT	set T-bit
SHAD Rm, Rn	shift arithmetic dynamic
SHAL Rn	shift arithmetic left by 1
SHAR Rn	shift arithmetic right by 1
SHLD Rm, Rn	shift logical dynamic
SHLL Rn	shift logical left by 1
SHLL2 Rn	shift logical left by 2
SHLL8 Rn	shift logical left by 8
SHLL16 Rn	shift logical left by 16
SHLR Rn	shift logical right by 1
SHLR2 Rn	shift logical right by 2
SHLR8 Rn	shift logical right by 8
SHLR16 Rn	shift logical right by 16
STC GBR, Rn	store to register from GBR
STC.L GBR, @-Rn	store to memory from GBR with pre-decrement
STS FPSCR, Rn	store to register from FPSCR
STS.L FPSCR, @-Rn	store to memory from FPSCR with pre-decrement
STS FPUL, Rn	store to register from FPUL
STS.L FPUL, @-Rn	store to memory from FPUL with pre-decrement
STS MACH, Rn	store to register from MACH
STS.L MACH, @-Rn	store to memory from MACH with pre-decrement

Table 173: SHcompact instruction set summary

Instruction	Summary
STS MACL, Rn	store to register from MACL
STS.L MACL, @-Rn	store to memory from MACL with pre-decrement
STS PR, Rn	store to register from PR
STS.L PR, @-Rn	store to memory from PR with pre-decrement
SUB Rm, Rn	subtract
SUBC Rm, Rn	subtract with carry
SUBV Rm, Rn	subtract with underflow check
SWAP.B Rm, Rn	swap register bytes
SWAP.W Rm, Rn	swap register words
TAS.B @Rn	test and set memory byte
TRAPA #imm	trap always
TST Rm, Rn	bitwise test, result placed in T-bit
TST #imm, R0	bitwise test immediate, result placed in T-bit
TST.B #imm, @(R0, GBR)	bitwise test memory, result placed in T-bit
XOR Rm, Rn	bitwise XOR
XOR #imm, R0	bitwise XOR immediate
XOR.B #imm, @(R0, GBR)	bitwise XOR memory
XTRCT Rm, Rn	extract a long-word

Table 173: SHcompact instruction set summary

SuperH

Index