SPARC-V8 Supplement

SPARC-V8 Embedded (V8E)
Architecture Specification

SPARC-V8E, Version 1.0
October 23, 1996

SPARC-V8E Release 1 Architecture Specification

Version 1.0

Attached is the SPARC-V8E Release 1 Architecture Specification, which supersedes SPARC-V8E D0.9 and all previous revisions.

Please destroy or appropriately archive any previous, out-of-date drafts of this document in your possession.

Please send any recommended additions, corrections or modifications to:

SPARC International
3333 Bowers Avenue, Suite 280
Santa Clara, CA 95054-2913
USA

Phone: (+1) 408-748-9111
Fax: (+1) 408-748-9777

Contents

SECTION PAGE

Preface

Introduction

Welcome to SPARC-V8E, the real-time / embedded extension of SPARC, the most prevalent RISC architecture for general purpose computing. SPARC-V8E adds instructions for increased performance and fast interrupt response time, defines critical system features and provides a reference architecture to support real-time debugging.

SPARC-V8E is a microprocessor specification created by the SPARC Embedded Committee of EuroSPARC and reviewed by the SPARC Architecture Committee of SPARC International. SPARC-V8E is not a specific chip; it is an architectural specification that can be implemented as a microprocessor by anyone securing a license from SPARC International.

EuroSPARC is an open membership SPARC user group in Europe who counts among its members real-time / embedded computer makers, semiconductor designers and manufacturers, as well as software development tools and operating systems vendors. SPARC International is a consortium of computer makers with membership open to any company in the world. The SPARC Embedded Committee has been chartered to enable and support the use of SPARC as the embedded architecture of choice. The SPARC Architecture Committee is composed of voting members each of whom represents one of SPARC International's Executive Member companies.

General purpose architectures are normally evolved to anticipate increasing demands of applications as well as to take advantage of state-of-the-art technology. SPARC-V8 and V9 are good examples. The Embedded Committee of EuroSPARC has identified an additional direction, that of creating a chip architecture that can bring RISC research and experience to real-time / embedded systems at volume prices. By using SPARC-V8 architecture as the base for enhancement, the Embedded Committee of EuroSPARC allows both workstations and embedded systems to share the benefits of volume prices and ongoing research. And the simplicity of V8 implementation has already made it a preference for custom modular chips such as the ones being designed for the SMILE project, a major European investment The resulting real-time/embedded extension of the V8 architecture creates a processor with high performance suitable for operating systems ranging from Solaris (tm), down to fully predictable, high speed real-time operation on minimum-sized executives. It also ensures that this processor, unlike others that were dedicated only to real-time / embedded use, will be long lived because it will profit from all of the innovation and investment going into the SPARC chips for workstation use.

Learning from experience obtained from Fujitsu's "SPARClite" chips, "DIVIDE STEP" instructions have been introduced to minimize interrupt response time. Additional instructions such as SCAN and MAC are intended to provide hardware support for high performance execution. The MMU has been extended, e.g. it supports global context and optional protection of a smaller page size. System features have been added: input handlers, interrupt mechanisms, counters, timers, pulsers. Finally, a complete real-time debug architecture has been defined to support breakpointing, tracing and emulation.

Architecture compatibility for implementations, is based on the common denominator, the V8 Architecture definition and enhancements. Compliance with the specification can be obtained either via complete H/W implementation or by instruction emulation.

Audience for this Specification

The audience for this specification includes implementors of the architecture and developers of SPARC-V8E system software (simulators, compilers, debuggers, and operating systems, for example). Software developers who need to write SPARC-V8E software in assembly language will also find this information useful.

Where to Start?

If you are new to the SPARC architecture, read <u>The SPARC Architecture Manual, Version 8</u> for background. Then look into the subsequent sections and annexes of this document for more details in areas of interest to you.

If you are already familiar with SPARC-V8, you will want to review the list of new features listed below and in the next section, Scope.

Specification Contents

The first section, Scope, describes the overall content of the document, and its relationship to SPARC-V8E. Section 2, "Instructions," reviews the Instruction Set Architecture (ISA) Extensions and enhancements to ASI accessibility. Section 3, "Memory Management Unit," is a description of the Memory Management Unit (MMU), Section 4, "Traps", describes the single-vector trapping features, Section 5, "Peripheral Extensions", covers Timers, Counters, and Interrupt Facilities, and Section 6, "Diagnostic Facilities" includes instruction tracing, setting and using breakpoints, single stepping and emulation.
Annexes follow the sections and include the following: Annex A, "Programming Techniques", Annex B, "Alternative Window Usage Models", and Annex C, "Summary of Operation codes, ASI's and ASR's".

Acknowledgments

The members of the SPARC- V8 Embedded architecture committee, set up in December

1991, devoted a great deal of time describing and discussing the design of the SPARC-V8E architecture. Originators of various sections of the initial draft specification include: Frits Zandveld (Philips) - Secretary, MMU, Counters, Timers and Interrupts; Bruce McKeever (Fujitsu)- instructions, counters and timers, interrupt control, diagnostics; Anna Hedbrant (Ellemtel)- MMU; Patrik Strömblad (Ellemtel)- MMU; Yves Roumazeilles (SAGEM)- Counters and timers; Cesar Douady (MHS)- general comments and instructions; Rafael Guzman (TGI)- Chair; Alain Fanet (MHS)- Chair.

Additional contributors and reviewers include: Max Baron (Sun); Edmund Kelly (Sun)- MMU; Rudolf Usselman (S-I)- instructions; Les Kohn (Sun)- evaluation of bitfield proposals; David Weaver (Sun)- interrupts, counters, timers, and overall editing and review support.

Others who contributed either via the “Task Force” to get the committee started or through the SPARC-V8E Architecture Subcommittee to finalize and produce the final specification include: J.J. Whelan (S-I)- Chair; Bruce McKeever (Fujitsu); Craig Nelson (LSI); Edmund Kelly (Sun); Mike Rayfield (TI); and Dalibor Vrsalovic (SunSoft).

Final consolidation of draft material and technical editing was provided by Morris Enfield (Enmor Associates) on contract to SPARC International.

1 Scope

This supplementary specification defines a 32-bit enhanced SPARC-V8 architecture called SPARC-V8E that is upward-compatible with the existing 32-bit SPARC-V8 processor architecture. This specification includes, but is not limited to, the definition of the enhanced instruction set, ASI access, trap model, memory management unit, diagnostic facilities, and timers and counters. Specific implementations may selectively include one or more of the specified supplementary features and functions.

1.1 SPARC-V8E Attributes

SPARC-V8E is a CPU **instruction set architecture** (ISA) and a **set of facilities** to improve programmer control over processor behavior and to improve processor responsiveness to the outside world in the context of embedded applications. It is derived from SPARC-V8. Both architectures come from a reduced instruction set computer (RISC) lineage. As architectures, SPARC-V8E and SPARC-V8 provide a basis for a spectrum of chip and system implementations at a variety of price/performance points. SPARC-V8E may be employed in a range of applications, including most embedded applications such as real-time, process control, medical, imaging, digital telecommunications, local area networking (LAN), and other time-critical and dedicated or embedded scientific and commercial applications.

1.1.1 Design Goals

SPARC-V8E, as specified, is a platform for optimizing and standardizing software systems, diagnostic tools, and high-performance hardware implementations.

1.1.2 Architectural Enhancements

SPARC-V8E is derived from SPARC-V8, which in turn is derived from SPARC, which was formulated at Sun Microsystems in 1985. SPARC is based on the RISC I & II designs engineered at the University of California at Berkeley from 1980 through 1982. Enhancements have been made based on requirements for improved performance and lower cost of operation in time-critical and dedicated processing environments.

The architecture provides for a spectrum of input/output (I/O) and memory-management unit (MMU) sub-architectures. SPARC-V8E assumes that these elements are best defined by the specific requirements of particular systems. Note that they are invisible to nearly all user programs, and the interfaces to them can be limited to localized modules in an associated operating system.

1.2 SPARC-V8E Features

SPARC-V8E includes the following enhanced features:

— Divide Step Instruction:

> The Divide Step instruction, **DIVScc**, provides for implementation of an interruptible divide algorithm.

— Scan Instruction:

> The Scan instruction, **SCAN**, provides the capability for quickly locating the first bit set, cleared, or differing from the sign in a word. Such operations occur frequently in embedded systems, especially for scheduling and interrupt case detection.

— Multiply Accumulate Instruction:

> The Multiply Accumulate instruction, **MAC**, enhances e.g. (integer) fast Fourier transform operations.

— Alternate Window Pointer Register:

> The Alternate Window Pointer Register, **AWP**, helps reducing the amount of time a SPARC-V8E is not interruptible during register save operations.

— Partial Write Program Status Word:

> Write Program Status Word, **WRPSR**, with a special value for the rd field allows atomic setting and resetting of the ET field in the Program Status Word.

— Non-Privileged ASI Access:

> Allows LOAD and STORE from Alternate space instruction access to some ASI's in user mode.

— MMU:

Improvements have been made to the Reference MMU to provide enhanced functionality while retaining compatibility with the SPARC-V8 Reference MMU.

The improvements include software table walk, page protection down to 1k bytes, bypass of context number checking in the address translation phase, and TLB entry locking.

— Traps:

A facility supporting single vector trapping has been provided.

— Interrupt Handlers:

A set of features to shape and prioritize interrupt signals has been added.

— Timers/Counters/Pulsers:

Improved timer/counter/pulser facilities will be provided as part of the Peripheral Extensions expanded functionality.

— Diagnostic Facilities:

Initial diagnostic features have been provided based on Fujitsu's SPARCLite Debug Support Unit specification.

1.3 SPARC-V8E Definition

The SPARC Version 8 embedded architecture, SPARC-V8E, is defined by the sections and normative annexes of this document. A correct implementation of the architecture provides for execution of a program strictly according to the rules and algorithms specified in the sections and normative annexes. The informative annexes provide supplementary information such as programming tips, expected usage, and assembly language syntax. These annexes are not binding on an implementation or user of a SPARC-V8e system.

The Architecture Committee of SPARC International has sole responsibility for clarification of the definitions in this document.

1.4 SPARC-V8E Compliance

Compliance to this specification may be claimed only by implementations which have been submitted to SPARC International for testing and which have been issued a certificate of compliance. Testing and certification of SPARC-V8E compliance requires that the implementation also be tested and certified as SPARC-V8 compliant with the exception of SPARC-V8E functionality which differs from SPARC-V8.

A compliant implementation need not implement all of the features described in section 1.2 or Annex D of this document. Each of the features identified in section 1.2 or Annex D can be individually implemented and certified as compliant. In order for a feature to be so certified it must be implemented as defined in this document. Claims of compliance to this specification must have the form, “Compliant to SPARC-V8E, Release 1 <feature list>” where <feature list> is the list of features certified as tested to be compliant by SPARC International. Compliance to SPARC-V8E must not be claimed without the feature list.

Annex D of this document formally lists all features and their legal combinations.

Prior to compliance testing, a statement must be submitted to SPARC International for each implementation that:

— specifies the individual features and functions of this specification selected for implementation and to be tested for compliance

— specifies the implementation choice for all implementation dependencies

— specifies any subsetting of function as allowed by this document

This information becomes the property of SPARC International and may be released publicly as part of a list of compliant implementations.

2 Instructions

The following instructions and Non-Privileged ASI access have been added to SPARC-V8 in order to improve performance and provide additional functionality especially for embedded, time critical applications. Divide Step, Scan, and ASI Non-Privileged access may be included individually or in any combination for implementation in a SPARC-V8E implementation.

2.1 Divide Step

opcode	op3	operation
DIVScc	011101	Divide Step (and modify cc's)

Format (3):

31 30	29 … 25	24 … 19	18 … 14	13	12 … 5	4 … 0
10	rd	op3	rs1	i=0	*unused* (*zero*)	rs2
10	rd	op3	rs1	i=1	simm13	

Suggested Assembly Language Syntax	
divscc	reg_{rs1}, *reg_or_imm*, reg_{rd}

Description:

The DIVScc instruction performs one bit-cycle of a non-restoring, shift-before-add, signed or unsigned division. Initially, the more significant half of the dividend is in the Y register, the less significant 32 bits are in r[*rs1*]. The divisor is in r[*rs2*]. Subsequently, the more significant half of the partial remainder is in the Y register, the less significant half is in r[*rs1*], along with another quotient bit.

DIVScc operates as follows:

(1) The *true sign* is formed using the negative (n) and overflow (v) integer condition codes from the Processor Status Register.

True sign= PSR.n **xor** PSR.v .

(2) The *remainder* is formed by left shifting the Y register (initially the more significant word of the dividend) one bit, and setting the least significant bit of the remainder equal to the most significant bit of r[*rs1*] (initially the less significant word of the dividend).

(3) The *divisor* is r[*rs2*] if the *i* field is 0, or *simm13*, sign-extended to 32 bits, if the *i* field is 1.

(4) If *true sign* = 0 (+), the ALU computes *(remainder - divisor)*. If *true sign* = 1 (-), the ALU computes *(remainder + divisor)*.

(5) Carry-out from the ALU operation is noted as c0. The negative (n) condition code is set to bit 31 of the ALU result. The zero (z) condition code is set if the ALU result is 0 and the *true sign* equals Y[31], otherwise it is cleared.

(6) The *new true sign* is formed as (*true sign* **and not** Y[31]) **or (not** c0 **and** (*true sign* **or not** Y[31]))

(7) The overflow (v) condition code is formed as *new true sign* **xor** bit 31 of the ALU result. The carry (c) condition code is set to **not** *new true sign*. Y is set to the 32-bit ALU result. If r[*rd*] is not 0, then r[*rd*] is set to r[*rs1*], left shifted one bit with **not** *new true sign* (the new quotient bit) in the least significant bit position.

Note:

Usage of DIVScc in other algorithms than in a division algorithm is not advised. This same warning applies to MULScc in SPARC Version 8 and Version 9.

Register Manipulation Description:

Divide step performs one bit cycle of a non-restoring, shift-before-add, signed or unsigned division. It operates on a signed or unsigned dividend with an unsigned divisor. It uses standard condition code bits to carry true sign, remainder, and previous quotient bit information from one cycle to the next. Therefore, standard SPARC instructions are sufficient for correct initialization for signed or unsigned divide, eliminating the need for a special divide initialize instruction. Use of shift-before-add and the functional equivalent of 33rd- bit add/subtract maintains remainder and quotient in correct relative position with respect to their holding registers, eliminating the need for a special divide terminate instruction to shift the last quotient without shifting the last remainder.

For non-overflow divisions, the non-restoring division leaves a last partial remainder bounded by: absolute divisor - 1 (|divisor| - 1), and, - absolute divisor (-|divisor|). With true sign last partial remainder carried by standard condition code bits, standard SPARC instructions are sufficient to produce the correct remainder, eliminating the need for a special remainder correction instruction.

Note:

Expected use of divide step will have r[*rd*] = r[*rs1*]. A useful exception is the first divide step of 32 by 32 signed division, which preserves the original dividend for later testing by r[*rd*] = r[*rs1*].

Traps:

none

2.2 Scan

opcode	op3	operation
SCAN	101100	scan for first occurrence of ‘1’ or ‘0’ bit

Format (3):

31 30	29 25	24 19	18 14	13	12 5	4 0
10	rd	op3	rs1	i=0	*unused* (*zero*)	rs2
10	rd	op3	rs1	i=1	simm13	

Suggested Assembly Language Syntax	
scan	reg_{rs1}, *reg_or_imm*, reg_{rd}

Description:

The SCAN instruction can be used to return the location of the first bit in r[*rs1*] that differs from its most-significant bit or the location of the first 1 bit or first 0 bit of source register r[*rs1*].

SCAN works as follows:

(1) The value in r[*rs1*] is “**xored**” on a bit-wise basis with the mask obtained by shifting right by one bit and sign-extending the value in r[*rs2*] if the *i* field is 0, or with sign_ext(*simm13*) if the *i* field is 1.

(2) The number of the bit position of the first “1” in the result from (1) above is returned to the destination register r[*rd*]. Bit numbers range from 0 for the most significant bit to 31 for the least significant bit. A “1” in the most significant bit (MSB) position returns a value of 0, while the first “1” in the least significant bit (LSB) position returns a value of 31. If no bit is set (the two operands are identical), an implementation dependent unsigned value greater than or equal to 32 is written to r[*rd*].

Implementation notes:

Use of an unsigned value with bit 31 set (but in any case greater than or equal to 64) is recommended for use in new implementations. The opcode for Scan is op=2, op3= 2C. The Scan instruction conflicts with SPARC Version 9 opcode for MOVcc.

Programming Note:

For portability, software must perform unsigned comparisons with the result produced by SCAN, since SCAN may return a value with bit 31 set to ‘1’.

Traps:

none

2.3 Mac

opcode	op3	operation
UMAC	tbd	Multiply and accumulate unsigned
UMACcc	tbd	Multiply and accumulate unsigned and modify cc
SMAC	tbd	Multiply and accumulate signed
SMACcc	tbd	Multiply and accumulate signed and modify cc

Format (3):

31 30	29 … 25	24 … 19	18 … 14	13	12 … 5	4 … 0
tbd	rd	op3	rs1	i=0	*unused* (*zero*)	rs2
tbd	rd	op3	rs1	i=1	simm13	

Suggested Assembly Language Syntax	
umac	reg_{rs1}, *reg_or_imm*, reg_{rd}
umaccc	reg_{rs1}, *reg_or_imm*, reg_{rd}
smac	reg_{rs1}, *reg_or_imm*, reg_{rd}
smaccc	reg_{rs1}, *reg_or_imm*, reg_{rd}

Description:

These instructions use ASRxx (xx tbd), Y and ASRyy (yy tbd) as an accumulator, ASRxx being the most significant word, Y the middle word and ASRyy the least significant word.

The product of both operands is computed as for the corresponding MUL instructions, but the result is operand1*operand2+ASRxx|Y|ASRyy. This result is stored in ASRxx (most significant word), Y (middle word) and both r[rd] and ASRyy (least significant word).

The width of ASRxx is implementation dependant. Its size can be observed from the software by writing full 1's in it and reading back to see set bits. In particular, ASRxx may contains no bits at all, in which case RDASRxx will return 0.

UMAC and SMAC do not affect the condition code bits. UMACcc and SMACcc set the condition codes the following way:

N, Z : As specified in V8, but replacing "product" by "result" (i.e. the accumulated result).

V, C : The condition out of the final addition, i.e. the result is computed on 1 more bit than the width of the accumulator and :

C is set if the extra bit is set and the operation is unsigned.

V is set if the extra bit is different from the most significant bit of the accumulator and the operation is signed.

ASRzz (zz tbd) contains 2 bits :

AccruedOverflow : Bit 1, set each time V=1.

AccruedCarry : Bit 0, set each time C=1.

ASRzz is only reset by writing to it.

ASRzz is updated even for instructions which do not affect condition codes .

The MUL instructions also sets ASRyy to the same value as r[rd] and ASRxx to 0 or full 1's depending on the result sign. Note that MUL is equivalent to ASRxx|Y|ASRyy=0 followed by MAC .

Traps:

none

2.4 Alternate Window Pointer

Description:

AWP (Alternate Window Pointer) is a field of ASRxx (no relation with previous section).

PSR contains two additional bits AW (Alternate Window, place tbd) and PAW (Previous AW, place tbd) which are reset on RESET. The current window is the one pointed to by CWP when AW=0 and the one pointed to by AWP when AW=1.

When a trap is taken, in addition to the normal behavior, AW is copied to PAW and AW is reset. Upon execution of RETT, PAW is copied back to AW.

This mechanism allows routines which manipulate windows other than the current window (such as context switching routines) to run with ET=1 thus reducing the maximum interrupt latency.

2.5 Partial WRPSR

Description:

When a WRPSR instruction with a non null rd is executed, only some fields of PSR are written rather than all the defined fields of PSR.The mapping “rd => fields” is tbd. However:

- rd=0 => all fields written (for compatibility)
- rd=tbd => only ET is written.

The second point allows to overcome the explicitly stated weakness of V8 (programming note 3 of the WRPSR instruction):

> If traps are enabled (ET=1), care must be taken if software is to disable them (ET=0) since the “RDPSR, WRPSR” sequence is interruptible - allowing PSR to be changed between the two instructions - this sequence is not a reliable mechanism to disable traps.

2.6 Non-Privileged ASI Access

In SPARC-V8E implementations providing for non-privileged ASI access functions, LOAD and STORE from Alternate space instructions accessing ASI’s 00_{16} - $7F_{16}$ are privileged instructions. LOAD and STORE from Alternate space instructions accessing ASI’s 80_{16} - FF_{16} are non-privileged instructions.

3 Memory Management Unit (MMU)

3.1 Overview

This specification describes a reference MMU for SPARC-V8E and a simple feature to indicate cacheability for those cases where no full reference MMU is to be implemented: a minimal MMU. The SPARC-V8E reference MMU is an extension to the existing reference MMU as described in the SPARC-V8 Architecture Specification. This specification covers the enhancements and modifications to the existing SPARC-V8 reference MMU to support embedded applications. It assumes an understanding of the architecture of the SPARC-V8 reference MMU.

The enhanced features and functionality offered by an embedded SPARC-V8E reference MMU are covered in Section 3.2 below. They include:

- — Sub-page protection down to 1k byte level
- — Support for disabling of context number match
- — Support for software table walk
- — Support for locking TLB entries

The Minimal MMU cacheability control can be provided in the case where no actual MMU is to be implemented.

3.2 Reference MMU architecture

3.2.1 Overview

The Embedded SPARC-V8E MMU architecture has been enhanced and/or modified in four principal areas:

(1) Memory protection has been extended down to a level of 1k bytes. This is done by splitting 4k byte pages into four subpages and providing protection for each of the four subpages. However, pages are always aligned to 4k byte boundaries, and, 4k bytes remains the minimum page size that may be addressed.

(2) Context Number matching in the Page Descriptor Cache (a.k.a. Translation Lookaside Buffer, or TLB) has been provided with an optional bypass. This provides for establishing global pages (as opposed to only local) that are accessible across process contexts.

(3) Support has been provided for a software table walk in addition to the hardware table walk of the SPARC-V8 MMU.

(4) Support has been provided for locking TLB entries.

Each of these enhancements are described in the following sections as specific modifications to the existing SPARC-V8 reference MMU. No attempt has been made to provide a full description of the existing SPARC-V8 reference MMU, however, some portions of the V8 reference MMU architecture are replicated here for background clarification of the Embedded V8e enhancements.

Note:

The SPARC-V8E MMU Specification uses the term TLB interchangeably with the term PDC (as currently used in the V8 reference MMU Specification).

3.2.2 Virtual Address Format

As defined in the reference MMU, the 32 bit address is subdivided into the following fields:

Index 1	Index 2	Index 3		Page offset
31 24	23 18	17 12	11 10	9 0

The lower 12 bits are used as an offset within the physical page.

Two bits of the page offset may be used during the comparison phase to provide protection for 1k byte pages. However, addressing itself is not modified and remains based on a minimum of 4k byte pages (see physical address below).

The three index fields correspond to lookup keys into three different translation structures, mapping 4k, 256k, 16M, or 4G of virtual addresses.

Implementation Note:

When only software table walk is supported, then the above format does not have to be followed in full and the detailed structure of the tables and their contents are purely a software matter. If hardware table walk is supported, table structures and table elements are as specified in a section below.

3.2.3 Physical Address Format

The physical address is a 36 bit field:

Physical page number (PPN)	Page offset
35 — 12	11 — 0

Note that the lower 12 bits of the physical address are the same as the lower 12 bits of the virtual address: they are not translated. This allows the implementation of virtually addressed, physically tagged caches with set sizes up to 4k bytes.

Implementation Note:

Implementation of all 36 physical address bits is not required.

3.2.4 Address Translation

The virtual address along with the context number are compared with the virtual address tags stored in the TLB. A match indicates that the translation from virtual to physical address is already in the TLB.

When a miss occurs, system hardware and/or software (See section on Hardware and Software Table Walk), will cause a trap that fetches the required PTE from the structures in memory. Due to sparse population and the use of large linear mappings, a full set of structures in most cases is not needed.

Access permissions are checked by hardware for each translation. If the requested access violates those permissions, a fault is generated and the appropriate status information is stored in the Fault Status Register and the Fault Address Register.

3.2.5 Contexts

Each virtual address is associated with a “context” number. The management of context numbers is the responsibility of the memory management software. The context number of the current running process is stored in the context register. In this architecture, the context number has one purpose:

> By comparing the context number in the TLB entry field with the virtual address context, memory protection between different processes is provided during address translation.

The context number can also be used as an index into a list of translation table structures.

Implementation Note:

> The range of context numbers is implementation dependent, but can not be greater than 0...127.

The primary difference between SPARC-V8E and the SPARC-V8 reference MMU, with respect to context number matching, is that the context number match can be disabled in SPARC-V8E. When disabled, the comparison of context number is not performed during address translation.

3.2.6 Tables

Elements of the V8e MMU table structure and contents have been enhanced to provide for:

- — protection down to 1k byte pages
- — support for disabling context number matching on address translation; this provides support for both "local" and "global" pages

The following diagram shows the hardware table walk for a matching virtual address. I3 (6 bits) may be extended by two bits (total 8 bits) from the VA page offset to provide 256 PTE Level-3 Table entries for 1k byte page protection.

.

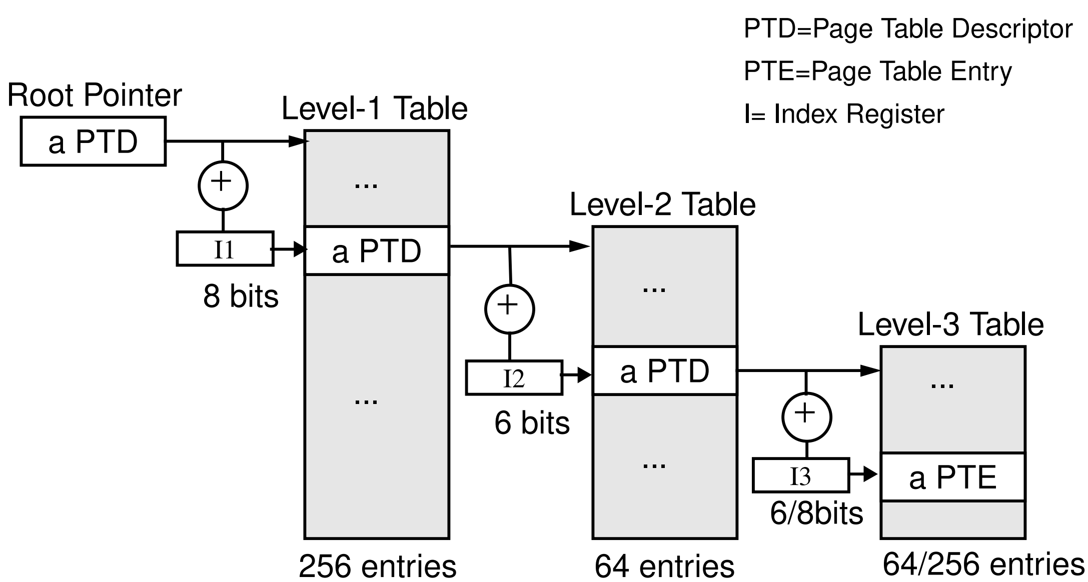

Figure 1: Page Table Search

The root pointer is unique to each context. It is found in the Context Table (see SPARC-V8 reference MMU description).

3.2.7 Page Table Descriptor (PTD)

The Page Table Descriptor is shown below. It has been enhanced to include a 1k byte subpage protection enable flag:

PTD:

PTP	R	KE	ET
31–4	3	2	1–0

PTP= Page Table Pointer value:
The PTP appears on bits 35 through 8 of the physical address bus during miss processing. The page table pointed to by a PTP must be aligned on a boundary equal to the size of the page table. The sizes of the three levels of page tables are the same as in the SPARC-V8 MMU

R = *Reserved*

KE= 1k byte protection enable (only at level 2):

KE = 0: I3 provides 6 bits- 64 page table entries;
page offset provides 12 bits- 4k bytes per page

KE= 1: I3 provides 8 bits- 256 page table entries;
page offset provides 10 bits- 1k bytes per subpage

ET = 01
Other entry types:
00: invalid
10: valid PTE (see PTE below)
11: valid PTE (see PTE below)

3.2.8 Page Table Entry (PTE)

The PTE has been enhanced in SPARC-V8E to support bypassing context number matching on address translation. This optionally provides for two types of pages:

— local pages, local to a particular context

— global pages, pages shared between contexts

PTE:

PPN	CMR	ACC	ET
31 – 8	7 – 5	4 – 2	1 – 0

PPN = Physical Page Number value:

The high-order 24 bits of the 36-bit physical address of the sub-page. The PPN appears on bits 35 through 12 of the physical address bus when translation completes

CMR = C,M and R bits as in V8 reference MMU

ACC = ACC bits as in V8 reference MMU

ET = 10: valid PTE for “local” subpage: perform the context number check
11: valid PTE for “global” subpage: do not perform the context number check
Other entry types:
00: invalid PTD
01: valid PTD

3.2.9 Translation Lookaside Buffer (TLB)

Miss processing of the TLB on virtual address translation may be provided by either hardware or software mechanisms or a combination of both. As previously noted, the TLB is referred to as a Page Descriptor Cache (PDC) in the V8 reference MMU specification. The terminology has been updated in this supplement to be consistent with the SPARC-V9 specification and industry convention.

3.2.9.1 Hardware and Software Table Walk

(1) <u>Software Table Walk</u>:

Software handling of miss processing uses an openly defined table organization and layout for the TLB. Details on loading a TLB element are specified in the sections below on “Writing TLB Entries”.

(2) <u>Hardware Table Walk</u>:

In the case of hardware miss processing of the TLB during virtual address translation, the software user is still required to know how to prepare the tables to be used by hardware address translation and table walk. However, the exact format of the transfer by hardware from these tables to a TLB is transparent to the software implementor.

(3) Hardware and Software Table Walk:

Even in the case of exclusive hardware miss processing, there exist requirements for software visible interfaces since software must initially construct the tables. For example, locking of TLB elements can only be provided by software, even when the table walk is in hardware. Consequently such functions as entering a lock bit into the hardware are detailed in the specification below. Moreover, even in a fully hardware tablewalk environment, reading and writing of TLB elements by software for diagnostic purposes may be useful along with other functionality.

3.2.9.2 TLB Contents

TLB entries are specified for software miss processing and other software access to the TLB. The TLB entry consists of two parts, an associative and a data part. The associative part is used during comparison matching with the virtual address. If an entry matches the virtual page address, then a physical page number (PPN) is directly provided by the data part of the TLB to generate the physical address. The TLB is composed of the following fields:

- PPN: Physical Page Number (up to 24 bits, implementation defined)
- C, M, R and ACC bits (as in V8)
- OL: Offset Length Indicator:
11: use 12 bits offset (1k subpages or 4k byte pages)
10: use 18 bits offset (256k byte pages)
01: use 24 bits offset (16M byte pages)
00: use 32 bits offset (4G byte pages
- VPN: Virtual Page Number, 20 virtual address bits comprised of Indexes: I1, I2, and I3
- K: 1k byte subpage identifier - equals two most significant bits of untranslated 12 bit VA page offset
- CN: Context Number
- KE: 1k byte subpage protection indicator:
KE =1: 1k byte subpage protection is enabled. Match 20 bits of VPN plus 2 bits from K (total 22 bits)
KE = 0: 4k byte page. Match only 20 bits of VPN
- V: Valid bit,
V=1: valid entry.
- G: Global bit, enabling switch for context field,
G = 1: do not check Context Number (global page)
G = 0: do compare Context Number (local page)
- TLB- Lock Bit: Locks the TLB entry- not to be changed by tablewalk hardware; can only be handled by software.

3.2.9.3 Address T ranslation

The following diagram illustrates the comparison and matc hing of TLB fields during translation of a virtual address into a physical address:

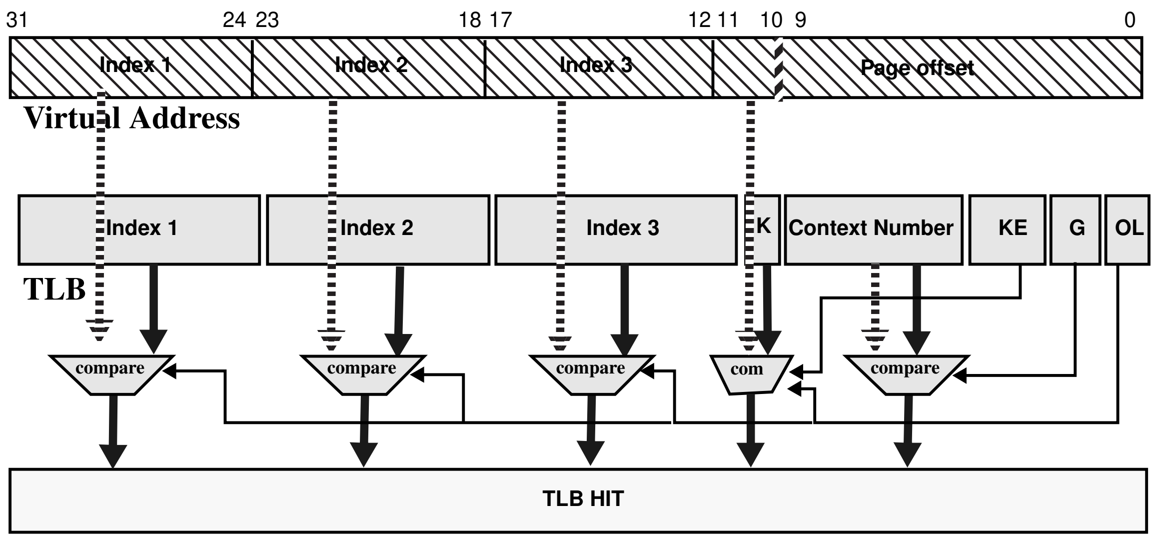

Figure 2: TLB Address T ranslation

3.2.9.4 Writing TLB Entries-Hardware T able W alk

When hardw are miss processing is implemented, the required data elements can be derived from the following sources:

element	Source
PPN	from the PTE being loaded
C,M,R,ACC	from the PTE being loaded
OL (offset length ind)	From table walk
VPN-virt page number	Virtual Address
CN-Context Number	Context Register
KE- protect enable	PTD- Level 2 (last read)
V-Valid Bit	Set to 1 if Table Walk ok (otherwise not entered)
G-Global Bit	from the PTE being loaded (least significant bit of ET)
LB-Lock Bit	Only manipulated by Software (can not be changed by hardware)

Table 1: TLB Entry Sources-H/W Table Walk

3.2.9.5 Writing TLB Entries- Software Table Walk

When software miss processing is implemented, the required TLB elements are available as a set of page descriptors located in a 4k byte area in an implementation defined location in ASI space (see TLB Mapping below).

16 bytes are allocated to each page descriptor, hence a maximum of 256 page descriptors can be supported. Page descriptor *n* is mapped on the 4 word area starting at byte address 16x *n* within the 4k byte space.

(1) Word 0 of a page descriptor is the physical address page descriptor word; it contains (in ASI address space):

Page Descriptor Word 0: Physical Address Data and Control Bits

PPN	CMR	ACC	OL
31 – 8	7 – 5	4 – 2	1 – 0

(2) Word 1 of a page descriptor is the TLB page descriptor word; It contains:

Page Descriptor Word 1: VA Tag and Control Bits

VPN	•K	Context Number	KE	V	G
31 – 12	11 – 10	9 – 3	2	1	0

(3) Word 2 of a page descriptor contains the Lock Bit;

Page Descriptor Word 2: Lock Bit

	LB
31	0

(4) Word 3 of a page descriptor is not used.

Page Descriptor Word 3: NA

Not Used
31 – 0

3.2.9.6 TLB Mapping

The following is an example of a reference implementation of MMU features for page descriptor mapping within the implementation dependent alternate address space .

Within this address space , TLB page descriptors may be accessed as shown in the following table:

Word	No. Bytes	Address
0	4	16*n* to 16*n*+3
1	4	16*n*+4 to 16*n*+7
2	4	16*n*+8 to 16*n*+11
3	4	16*n*+12 to 16*n*+15- Not Used

Table 2: TLB Page Descriptor Mapping

Implementation note:

> For software portability across implementations , 1k byte subpages are assumed to be referenced by the same Physical Page Number (PPN). However , this does not preclude individual implementations with 1k byte mapping .

3.3 Cacheability Control (Minimal MMU)

If the reference MMU is not implemented, it may be desirable to control caching of memory accesses , e.g. to prevent caching data from blocks where DMA is in progress . In such cases , the following function may be implemented:

> When the most significant bit (MSB) of an instruction address or data address equals one , the item referenced is not cacheable . The remaining 31 virtual address bits are used, without translation, as physical address bits .

4 T raps

4.1 Overview

This section contains enhancements to the SPARC-V8 trap specifications. Vectoring all traps through a single vector, single-vector trapping, can improve performance and memory utilization if all trap service routines can fit into cache memory.

4.2 Single-Vector Trapping

As an alternative to the standard SPARC-V8 trap mechanism, a single vector trap mechanism is provided in SPARC-V8E. When this mechanism is implemented:

- trap type = 0: reset- vectors to a fixed physical address , $0x0$
- trap type > 0: all other traps- vector to Trap Base Address $+ 0$

After a trap has been taken, its Trap Type can be determined by reading the Trap Type field, TT, of the Trap Base Register (TBR). This can be used by software to determine subsequent processing of the trap. The trap base register has the same layout as in SPARC-V8:

Trap Base Address (high order 20 bits)	Trap Type (tt)	0 0 0 0
31 – 12	11 – 4	3 – 0

Trap Base Register

Single vector trapping can save code space and improve the response time of traps, since the most frequent trap service routines for a given application may fit and be locked in cache as needed.

Single vector trapping is enabled by setting the SVT flag, bit 0 of ASR 17, to a 1. A reset trap clears the SVT flag, making V8e implementations consistent with the SPARC-V8 specification.

All other trap features are as specified in the SPARC-V8 specification and the reader is referred to that document for their detailed description.

5 P eripheral Extensions

5.1 Overview

This section supplements the SPARC-V8 features and functions in areas peripheral to the basic processor such as: input handlers, interrupt mechanisms, timers, counters and pulsers. The peripheral extensions may be included individually in specific implementations.

The input handler described in 5.2 can be used to shape, buffer, mask and reduce noise on any inputs. Control of the features (polarity, noise immunity, buffering, masking) is controlled on an input - by - input basis: one register controls all such features for one input.

The interrupt prioritizer as in 5.3 makes use of input handlers as in 5.2; it furthermore merely prioritizes interrupts in one or more levels.

The integrated interrupt request controller as in 5.4 combines functions comparable to those of of input handlers as in 5.2 and prioritizer as in 5.3; furthermore control is on a function - by - function basis: one register controls polarity and noise reduction for all inputs; one register controls all masks, etc. This difference in control philosophy reflects two sets of user desires.

The counter-timer-pulser as in 5.5 standardizes a rather simple but nevertheless versatile counter, prescaler and counter output control construction.

The simple counters as in 5.6 support routine capabilities such as DRAM refresh signaling. The simple timers as in 5.6 support more demanding tasks such as periodic interrupt, simple and watchdog timeout signaling and square wave generation.

5.2 Input Handler

A generalized Input Handler for SPARC-V8E is specified. An Input Handler can drive the interrupt handling circuitry or drive or control a timer or counter. Input signals are first handled by a standardized edge control, noise immunity control, buffer and enable circuit. Noise immunity is attained by synchronizing the input sample clock with the processor clock. The input sample clock may have the same period as the processor clock or it may be divided down. The ratio of processor clock to input sample clock is an implementation parameter. The number of samples taken can be controlled.

Control of each separate input line is done via one control register, controlling edge as well as number of samples, buffering and masking.

5.2.1 Input Handler Circuit

The following diagram shows the basic Input Handler Circuit:

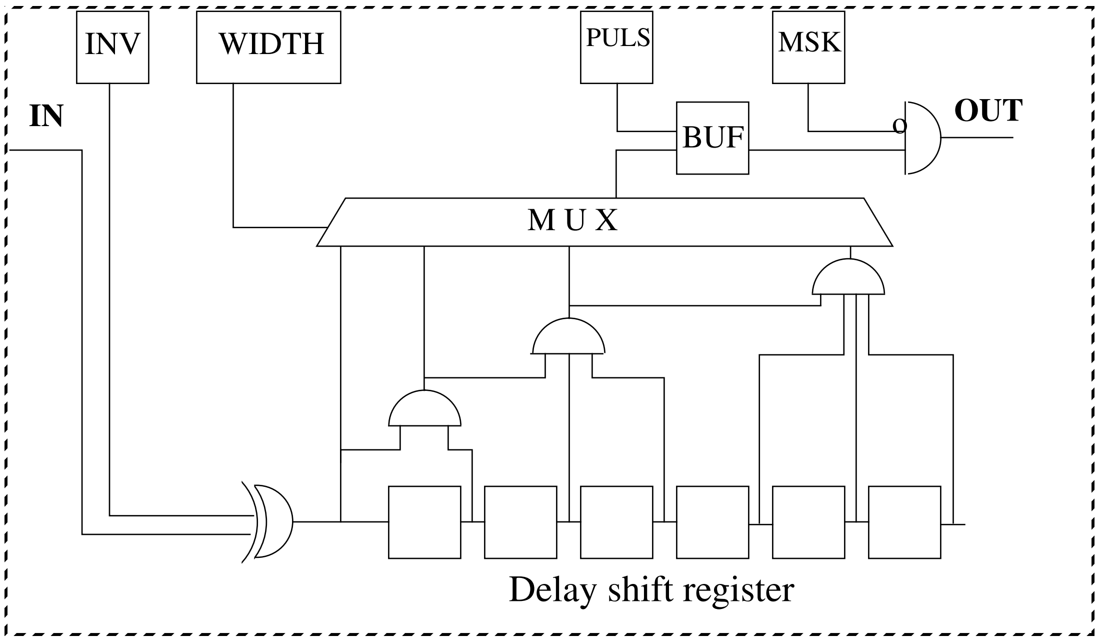

Figure 3: Input Handler Circuitry

- IN: the input signal to be handled
- INV: control signal **:**
  0 = do not invert IN
  1 = invert IN
- INP: IN **xor** INV
- SHIFT : a 6 bit shift register with INP as its input and shifted by the system c lock.
- WIDTH: control signal ***:***
  00 => set BUF to 1 when INP = 1
  01 => set BUF to 1 when INP = 1 **and** SHIFT[5] = 1
  10 => set BUF to 1 when INP = 1 **and** SHIFT[5:3] = 111
  11 => set BUF to 1 when INP = 1 **and** SHIFT[5:0] = 111111
- EN: control signal; whic h enables BUF to the output of the input handler .
- BUF: output signal; the latc hed result

Control signals INV, WIDTH, EN and the buffered result BUF can all be read and written by software. Reading will usually be done for diagnostic reasons only; BUF may be polled. IN, INP, and SHIFT can not be read or written by software. Each set of 5 signals (INV, WIDTH, EN and BUF) is mapped to one ASI word (see below).

Note:

An output should be disabled before its INV control or its WIDTH control are changed, otherwise an output signal change not reflecting an input signal change may occur. This mechanism may be used to produce a desired interrupt on a selected line.

5.2.2 ASI Mapping for Input Handler

All circuitry for one input is mapped on one ASI as follows:

— ASI 1_{16} or $C1_{16}$ is used for Input Handlers.

— Input Handlers are mapped in the 4k byte page starting at address 1000_{16} (4k bytes).

— each Input Handler is mapped onto a full word in the alternate address space (this permits up to 1024 input handlers). The bits are mapped as specified in the following table:

Bit	Description
32:5	unused
4	INV
3:2	WIDTH
1	EN
0	BUF

Table 3: Input Handler Mapping

Input handlers come in groups of 15. The outputs, OUT ($16n + 1$ through $16n + 15$), are mapped on the word at address $(16n + 0)*4$; this allows reading (polling) all outputs of a group and finding the leftmost ‘1’ using the SCAN instruction.

5.3 Interrupts

This section contains enhancements to the basic SPARC-V8 interrupt specification.

5.3.1 Overview:

The interrupt handler, when combined with a sufficient number of input mechanisms as described in 5.2, constructs a 15 channel programmable trigger input controller that arbitrates pending unmasked interrupt requests, encodes the highest priority interrupt request into a 4 bit code compatible with the SPARC-V8 defined Interrupt Request Level (IRL), and applies this code to IRL[3:0].

On top of this, a way is described to extend the number of interrupts handled over 15.

The following extensions are made to the SPARC-V8 interrupt specification:

— deriving the 4-bit IRL signal, as defined in V8, from (15) separate interrupt signals

— buffering of pulse-shaped interrupt signals

— establishing the polarity of interrupt signals

— supporting more than 15 interrupt sources

— masking interrupts

The basic mechanism contains up to 15 Input Handlers (as previously described). The Input Handlers are in turn connected to a priority circuit, thus generating a 4 bit code to be used as the Interrupt Request Level (IRL). The basic interrupt circuit is shown below.

5.3.2 The basic circuitry

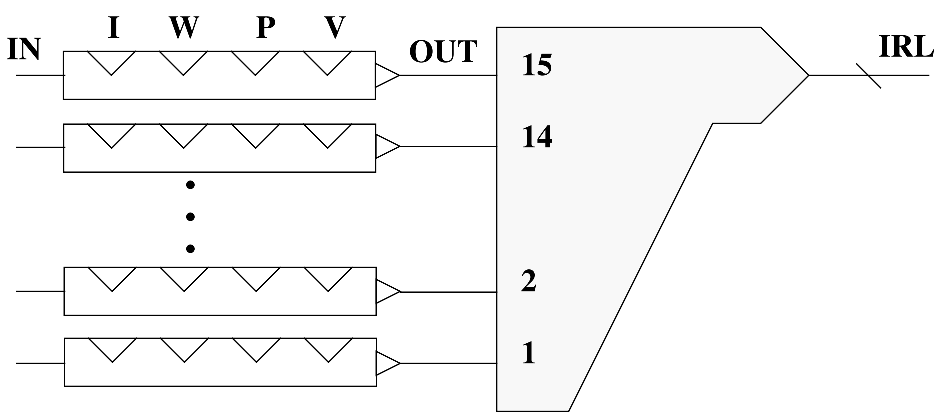

Figure 4: Basic Interrupt Mechanism

Each set of flip-flops, INV, BUF, WIDTH, and EN can be separately read and written by software. Reading will usually only be done for diagnostic (or polling) purposes. Resetting the actual input signals is not part of the interrupt mechanism.

Note:

An interrupt INT should be disabled before its INV or its WIDTH controls are changed, otherwise an output signal change not reflecting an input signal change may occur.

5.3.3 Extended Interrupt Mechanism

If more than 15 interrupts are to be provided, then the outputs of the Input Handlers can be grouped together by means of an **or** gate replacing the priority encoder in order to implement an extended interrupt mechanism. This extended interrupt mechanism is depicted below.

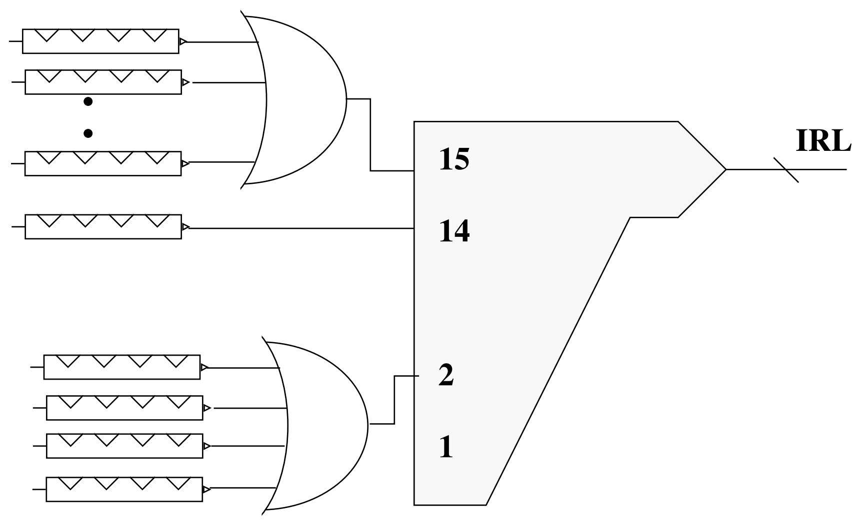

Figure 5: Extended Interrupt Mechanism (example configuration)

The exact interrupt within the inputs to such an **or** circuit must be detected by software scanning all BUF flip-flops driving those inputs. There is no limit to the number of “branches” in the extended circuit.

5.4 Integrated Interrupt Request Controller

5.4.1 Block Diagram and Overview

The Integrated Interrupt Request Controller(IIRC) is a 15 channel, programmable trigger interrupt controller that arbitrates pending unmasked interrupt requests, encodes the highest priority interrupt request into a 4 bit code compatible with SPARC-V8 defined Interrupt Request Level(IRL), and applies this code to IRL<3:0>. If traps are enabled and the code on IRL<3:0> is greater than the processor interrupt level set in the processor state register or the code is 15, then the processor is interrupted. The processor responds by servicing the interrupt and clearing the latched interrupt request in IIRC. Figure 6 shows a block diagram of the IIRC.

Control of the IIRC is on a feature-by-feature basis: all inputs share the input control register, the latch register and the mask register.

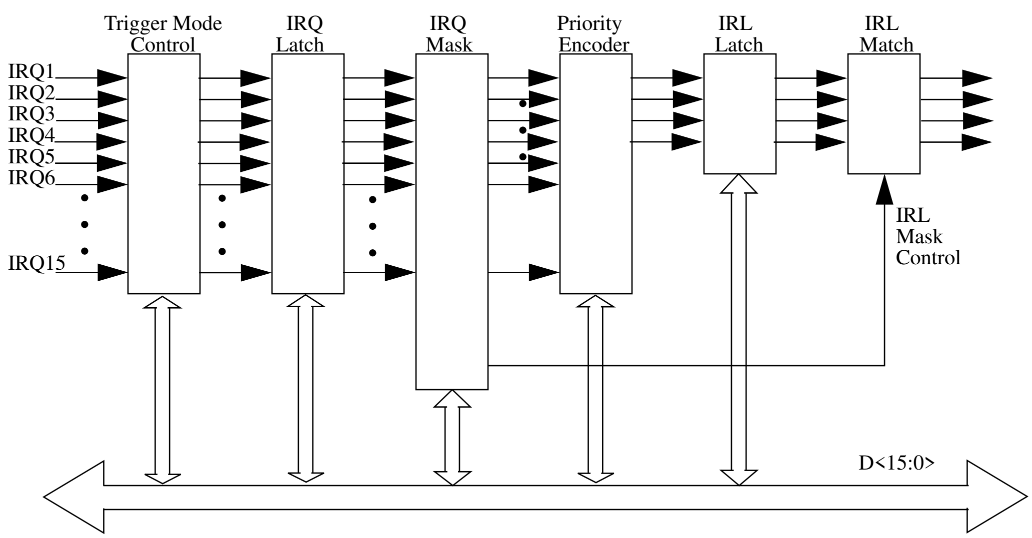

Figure 6: IIRC Block Diagram

Trigger Mode Control logic selects one of four trigger modes for each channel: high level, low level, rising edge or falling edge. The program sets the selection code by writing to the Trigger Mode registers.

Each Interrupt Request that satisfies the trigger mode conditions is captured in the IRQ latch. The program may read the latch through the Request Sense register and may clear the latch by writing to the Request Clear register.

Individual Interrupt Requests may be blocked by the IRQ Mask logic. The program controls the masking by writing to the Mask register.

All unmasked Interrupt Requests are examined by the Priority Encoder. The highest priority is encoded. IRQ15 has the highest priority and IRQ1 the lowest.

That encoded interrupt level from the Priority Encoder is captured in the IRL Latch.

The IRL Mask logic can block all interrupt requests by forcing the output of the IRL Latch going to the IRL lines to zero. The program controls IRL masking by writing to a reserved bit in the Mask register. Even if the IRL Latch is masked off, programs may poll for pending interrupts by reading the Request Sense register.

5.4.2 IIRC Registers

The IIRC has six internal registers that allow the program to control IIRC operation and to monitor interrupt requests that may be pending. Registers are mapped, aligned by function, into ASI 1 at successive word addresses as shown in Table X.

Address	Register	Required Access
IRC-REG + 0	Trigger Mode 0	Write
IIRC-REG + 4	Trigger Mode 1	Write
IIRC-REG + 8	Request Sense	Read
IIRC-REG + C	Request Clear	Write
IIRC-REG +10	Mask	Write
IIRC-REG +14	IRL Latch/Clear	Read/Write

Table 4: IIRC Register Memory Map

5.4.2.1 Trigger Modes Register Operation

Trigger Mode registers control the trigger mode for each interrupt channel. Trigger Mode Register 0 controls modes for channels 8-15. Trigger Mode Register 1 controls modes for channels 1-7.

reserved	md15	md14	md13	md12	md11	md10	md9	md8
31 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0

Trigger Mode Register 0

reserved	md 7	md 6	md 5	md 4	md 3	md 2	md 1	r
31 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0

Trigger Mode Register 1

Each two bit field in the registers selects one of four trigger modes for each channel as follows:

MDx	Trigger Mode
0	high level
1	low level
2	rising edge
3	falling edge

Table 5: Trigger Mode

Reset clears the Trigger Mode registers, initializing high level triggering for all interrupt channels.

Trigger modes will be explained in 5.4.3.3.

Note:

Changing trigger mode of an unmasked channel may result in a false interrupt.

5.4.2.2 Request Sense Register Operation

The program reads the state of the IRQ Latch through the Request Sense Register. Each one bit indicates a pending interrupt.

.

reserved	Request Sense	r
31 16	15 1	0

Request Sense Register

Reset clears the Request Sense Register.

5.4.2.3 Request Clear Register Operation

Writing one's to selected bits of the Request Clear Register clears corresponding elements of the IRQ Latch. The program typically uses this register to clear the latch element associated with the interrupt channel that it has begun to service.

reserved	Request Clear	r
31 16	15 1	0

Request Clear Register

Reset clears the Request Clear Register.

Note:

When changing trigger mode for an interrupt channel, its IRQ Latch element may be set and if not cleared after the change in trigger mode, may result in a false interrupt.

5.4.2.4 Mask Register Operation

Ones in the Mask Register block corresponding outputs of the IRQ Latch from examination by Priority Encoder, or, alternatively with a one in bit zero of the Mask Register, block the output of the IRL Latch from driving the IRL<3:0> lines. Using the Mask Register, the program may mask unused interrupt channels, temporarily mask individual active interrupt channels or mask all interrupt channels.

reserved	Mask	MKIRL
31 16	15 1	0

Mask Register

Reset clears the Mask Register.

5.4.2.5 IRL Latch/Clear Register Operation

The program uses the IRL Latch/Clear register to read or clear the IRL Latch. Reading this register will return the IRL code on bits 3:0. Writing one to bit 4 of this register will clear the IRL Latch. These capabilities permit optionally handling interrupt requests by polling rather than vectored interrupts.

reserved	*reserved*	Cl	int. level
31 — 16	15 — 5	4	3 — 0

IRL Latch/Clear Register

Cl= clear
int. level= interrupt level

Reset clears the IRL Latch/Clear Register

5.4.3 IIRC Operation

The IIRC latches interrupt requests into the IRQ Latch according to the trigger mode option selected for each interrupt channel. The Priority Encoder prioritizes unmasked interrupts and generates an encoded interrupt level code for the highest priority interrupt. The IRL Latch holds that code which is transferred through the IRL Mask logic to the IRL<3:0> lines for processor interrupt. If an interrupt occurs, the response program services the interrupt request identified on IRL<3:0> and clears both the IRL Latch and the latched interrupt request in the IRQ Latch.

5.4.3.1 Polling

The processor can poll interrupts by reading either the IRQ Latch via the Request Sense register or the IRL Latch via the IRL Latch/Clear register.

The processor may mask interrupts that it polls via the Request Sense Register by masking individual elements of the IRQ Latch or by masking the entire IRL Latch. Typically the processor periodically reads the IRQ Latch and clears interrupts from the latch as they are serviced. In case of multiple entries in the IRQ Latch, the SCAN instruction can be used to identify the one entry in the highest bit position. The IRL Latch may remain unmasked to allow vectored interrupt servicing of some interrupt requests if polled interrupts are masked in the IRQ Latch field of the Mask Register so that they are blocked from the Priority Encoder.

The processor may mask all interrupts when it polls via the IRL Latch/Clear Register by masking the IRL Latch, Mask Register bit 0 = 1. Typically, the processor periodically reads the IRL Latch for the highest level pending interrupt and clears both the IRL Latch and the interrupt from the IRQ Latch once the interrupt is serviced.

5.4.3.2 Initialization

All IIRC registers are cleared to 0 by Reset. This results in high level trigger mode for all interrupts and all masks disabled.

After reset, the interrupt trigger should be changed after the interrupts are masked with the IRQ mask to eliminate false interrupts. Following this, the IRQ Latches should be cleared, then the masks can be disabled.

5.4.3.3 Noise Immunity

The interrupt sample clock is synchronized with the processor clock and it is used to examine IRL<3:0> and engage the trap mechanism. The interrupt sample clock may have the same period as the processor clock or may be divided down. The ratio of processor clock to interrupt sample clock is an implementation parameter. Sampling of the incoming interrupt request signals takes place at the rising or falling edge of the interrupt sample clock. This is also an implementation parameter.

For level mode triggering, a number of successive samples at the required level must occur. That number is also an implementation parameter.

For edge mode triggering, the signal must currently satisfy high level conditions for rising edge or low level conditions for falling edge. Additionally, prior to the signal satisfying the appropriate current conditions, it must have satisfied the opposite level condition for another number of successive samples. That second number is an implementation parameter. Figure 7 shows an example of level detection for processor to interrupt sample clock ratio = 2, sample on clock rising edge and number of successive clocks at required level = 2.

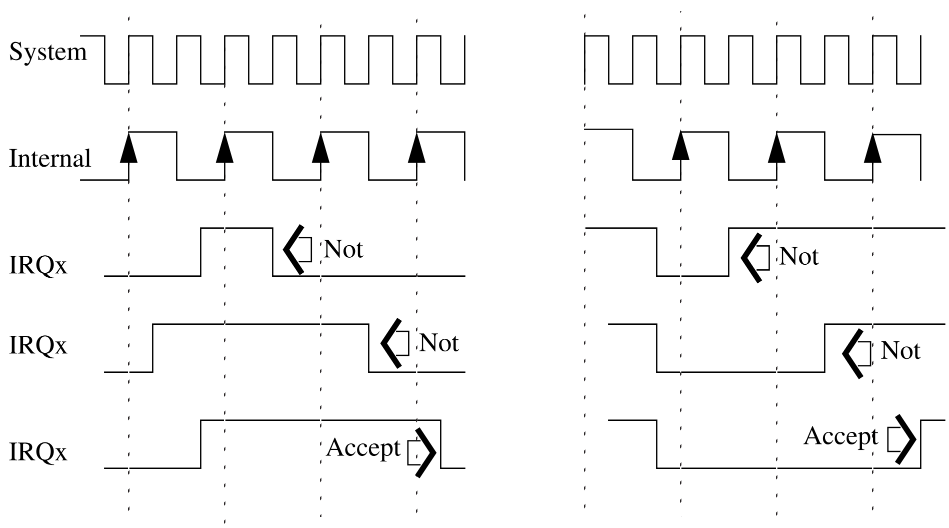

Figure 7: IRC Level Mode Trigger Sample Timing

Figure 8 shows an example of edge detection with the same implementation parameters plus number of prior opposite level samples = 1.

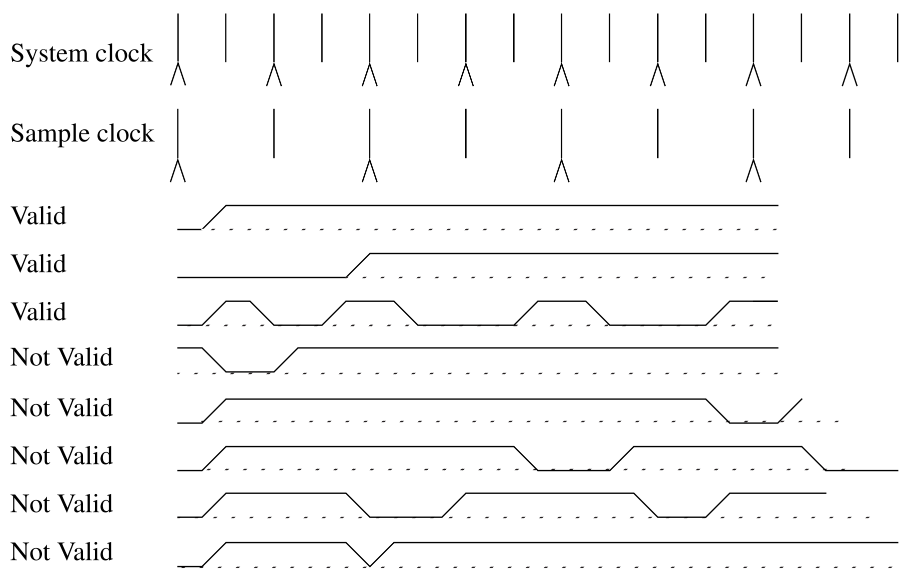

Figure 8: Edge Mode Trigger Sample Timing

5.4.4 Extension for Additional Interrupt Sources

SPARC-V8 provides for 15 interrupt request levels. If there are more than 15 interrupt sources, then a multi-stage interrupt processor must be constructed, with inputs of comparable priority being latched and grouped together into a single IRQ line of the Integrated Interrupt Request Controller. After interrupt servicing begins, the program must read the register of the grouped inputs and determine which one within the group has the highest priority. With appropriate conventions of bit position mapping, this can be done efficiently using the SCAN instruction.

5.4.4.1 Use of Input Handlers

Signals at a second or higher stage, that are grouped into a single first stage IRQ line, may be conditioned through Input Handlers as described in 5.2.1 and the grouping may be done through the **OR**ing as described in Section 5.3.3. When resolution of specific individual inputs to be serviced is required, the grouped Input Handler Outputs can be read and scanned.

5.4.4.2 Use of Externally Latched and Buffered Signals

Signals at a second or higher stage, that are grouped into a single first stage IRQ line, may be conditioned through a reduced IIRC. This consists of the Trigger Mode Control, IRQ Latch and IRQ Mask. The 15 outputs are **OR**ed to a single value which is masked by bit 0 of the IRQ Mask and then connected to a single group input. This is a single first stage IRQ line or input to another stage. When resolution of specific individual inputs to be serviced is required, the grouped Input Handler Outputs can be read and scanned.

5.5 Timers and Counters

5.5.1 Programmable Pulse Generators

5.5.1.1 Overview

The following extended timer and counter features and functions are specified for SPARC-V8E. This specification provides, in addition to a generalized counter mechanism, a two stage counter mechanism for setting the step size in which actual time counts are performed. It also provides increased flexibility in implementing various types of signals generated when a counter overflows.

SPARC-V8E timers and counters are designed to deliver a pulse of a certain shape to be used, for example, as an interrupt. Hence, a timer/counter is actually a pulse generator that produces a pulse after a specified delay.

For example, a pulse generator is required for slow I/O (as in actuator signals and slow serial output ports) or to reset an interrupt source, regardless of whether it originates on- or off-chip. This specification describes a bank of general purpose timer/counter pulse generators that may be employed in varying implementation contexts.

This specification does not stipulate what is to be counted, as that is left to specific implementations of the general mechanism(s). Examples of elements that can be implemented include:

— processor clock pulses

— processor clock pulses divided by *n* (where *n*=16, for example)

— input pulses for an on-chip device or an off-chip device (possibly handled first by an input handler as specified in the section on Input Handlers).

This specification does not stipulate the destinations for generated counter/timer pulse values as that is left to specific implementations. Examples of generated pulse destinations include:

— interrupt line(s) (see interrupt specification)

— on-chip device(s)

— off-chip device(s) (output via a chip output pin)

5.5.1.2 Timer/Counter Mapping

A bank of *n* general purpose timer/counter pulse generators is defined. Each timer, counter, and generator consists of three parts:

— (step-) pre-SCALER

— (step-) COUNTER

— (pulse-) SHAPER

The timer/counter pulse generators are mapped on an alternate address space (ASI 1_{16} or 81_{16}) (see below). A bank of 4096 addresses is reserved for timer/counters. Each timer/counter occupies 32 bytes within this bank, although not every byte is used. A maximum of 128 timer/counter pulse generators can be mapped.

5.5.1.3 SCALER

SCALER is a counter to adjust the value by which COUNTER is incremented. It is the first stage of a two-stage general counting mechanism. An important example of its usage is to compensate for differences in clock frequency among various implementations of SPARC-V8E. The SCALER counts processor clock cycles (or other inputs to the counting circuitry, such as clock cycle/16).

For example, for COUNTER to count in milliseconds in an application where the clock frequency is 100 MHz, SCALER would be set to 100,000 (or to 100,000/16 when clock cycle/16 is used as input to the counting circuitry). Frequently, all SCALERs will hold the same value (e.g. corresponding to 1 msec) which can be written at system start-up time and never changed. Software executing after SCALER is set could then be written to be portable across SPARC-V8E implementations.

SCALER can be controlled by two “external” signals, each provided by an input handler (see section on Input Handlers).

SCALER consists of *s*-bit registers, SCALER.set and SCALER.cnt. *s* is implementation dependent but within the range of 8 to 32, inclusive.

SCALER.cnt decrements at every count pulse (in a particular reference implementation, the actual count pulse is an external pulse handled by an input handler; refer to section on Input Handler).

SCALER has an Enable input signal whose particular reference implementation is typically provided by an input handler. Counting is enabled when the Enable signal=1. If the Enable signal is driven by the output of a signal handler:

- counting can be controlled by an external signal after handling by the INV, WIDTH, BUF and EN construct (see Input Handler)
- counting can also be controlled by software:

  -count when BUF=1 and EN=1

  -stop when BUF=0 or EN=0

When SCALER.cnt reaches zero (as controlled by SHAPER) it either:

- stops counting or
- copies SCALER.set and continues decrementing

Which action is taken depends on the associated SHAPER value. When SCALER.cnt continues after reaching zero, it sends one count pulse to COUNT. The counter mechanism is started by writing a 1 into the SCALER copy bit (see Shaper below). SCALER.cnt then copies SCALER.set and starts counting

5.5.1.4 COUNTER

COUNTER is a counter to actually count in steps as set by the value in SCALER. COUNTER is decremented when SCALER.cnt reaches zero.

COUNTER consists of:

— Two *c*-bit registers, COUNTER.set and COUNTER.cnt.

  c is implementation dependent and may vary from 8 to 32 bits.

COUNTER operates as follows:

— COUNTER.cnt and COUNTER.set are written simultaneously with the same start value

— COUNTER.cnt can be read; Readability of COUNTER.set is not required

— COUNTER.cnt is decremented by 1 when SCALER.cnt reaches zero

— When COUNTER.cnt reaches zero (as controlled by SHAPER, see section 5.4.5 below):

- Counting stops, or
- The value in COUNTER.set is copied into COUNTER.cnt and counting continues.

5.5.1.5 SHAPER

When the Counter.cnt reaches zero, the pulse generated may take one of various forms, including:

— a Positive Step

— a Negative Pulse

— Pulse after delay

— Pulse of specified width

— Series of pulses as depicted in the following figure

:

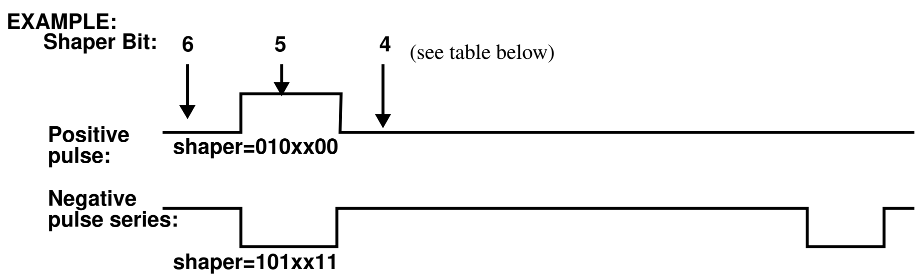

Figure 9: SHAPER Pulse Generation

The SHAPER controls a generalized pulse generation facility. The SHAPER is implemented as an 8-bit register. The SHAPER register values are mapped on 8 bits as follows (bit 0 is the least significant bit)

Note: Bits 6:4 of the SHAPER control the values of the output signal;
Bits 3:0 of the SHAPER control counters stopping, starting and continuing.

This is explained in the table below:

:

Bit	Description
31:7	Undefined (may be used for extensions)
6	Start value of the output signal: The value of the output signal when COUNTER is started by writing a 1 to the start bit-2
5	Value of the output signal when COUNTER reaches zero for the first time after restart
4	Value of the output signal when SCALER reaches zero for the first time after COUNTER reached 0
3	Reserved
2	When value is written to 1, starts the counter by copying SCALER.set into SCALER.cnt and COUNTER.set into COUNTER.cnt and outputting a signal- bit 6. This is used to start a prepared counter at a precise instant. The degree of precision is as precise as the input (which e.g. may be clock/16)
1	Bit 1=0: when COUNTER reaches 0, stop COUNTER. Bit 1=1: when COUNTER reaches 0, copy COUNTER.set and restart counting.
0	Bit 0=0: when SCALER and COUNTER reach 0, stop SCALER. Bit 0=1: when SCALER and COUNTER reach 0, copy SCALER.set; restart counting.

Table 6: Shaper Register Mapping

5.5.1.6 ASI Mapping for Counters and Timers

When mapping counters and timers on an alternate address space, it is recommended that ASI 1_{16} or 81_{16} be used. Addresses aligned on 4k byte blocks are available, providing 128 counter/timer/pulsers:

— COUNTER.set[*n*] is the word at address: 32*n* to 32*n*+3

— COUNTER.cnt[*n*] is the word at address: 32*n*+4 to 32*n*+7

— SCALER.set[*n*] is the word at address: 32*n*+8 to 32*n*+11

— SCALER.cnt[*n*] is the word at address: 32*n*+12 to 32*n*+15

— SHAPER[*n*] is the word at address: 32*n*+16

Note:

Elements containing less than 32 bits are mapped on the least significant bits (LSB) of these words. The remaining higher order bits in the word are unused.

5.5.1.7 Examples

A continuous timer with a large capacity can be mapped.

In this example:

— set SCALER to FFFF $FFFF_{16}$

— set COUNTER to FFFF $FFFF_{16}$

— set SHAPER to *x*01*x* *x*1*xx*2 (*x*= don’t care)

— when:

- SCALER is 16 bits (FFFF) and
- COUNTER is 24 bits (FF FFFF) and
- input is clock/16 then

COUNTER concatenated with SCALER will show FF FFFF $FFFF_{16}$ minus the number of input pulses since it was set.

COUNTER, concatenated with SCALER, has, at 100 MHz processor clock frequency, a “capacity” of 16 x 2^{16} x 2^{24}/ 100M = 176k sec = 49 hr = just over 2 days.

An interval or watchdog timer with an interval of X msec.

This can be implemented in many ways. One example follows:

— set SCALER to 100 microsec. (expressed in input pulses) at system start-up. Do not change after start-up

— set COUNTER to 10 x *X*

— set SHAPER to *x*010 *x*111 (repeated “1” pulses, 100 microseconds wide) which indicates:

- pulse start value = 0 (see bit 6);
- when COUNTER reaches zero, Output pulse value = 1 and remains 1 until SCALER reaches zero (as indicated by bit 5):
- when SCALER reaches zero, output value = 0 and remains 0 until changed (as indicated by bit 4 being 0):
- repeat by restarting COUNTER and SCALER when they reach zero (as indicated by bits 1 and 0 being ‘112’

Time can be measured in milliseconds:

— set SCALER to 1 msec (expressed in input pulses). Do this at system start-up and do not change

— set COUNTER to FFFF FFFF $FFFF_{16}$

— set SHAPER to *xxxx x*1*xx* to start it

— start the event timer via the Enable input mechanism

Input handlers may be used to construct counters for counting any input. A counter constructed from a general input handler will count the applied input pulses. Such a counter may also be started and stopped by applying the input handler's EN bit.

5.5.2 Simple Counters

Simple counters ranging from 8 to 32 bits may be implemented to support routine capabilities. The number of bits and the number of counters is an implementation parameter.

The counters are driven by the processor clock and are mapped into memory at ASI 1 or $C1_{16}$. Each counter occupies a whole word address. Associated with each counter, at the next word address in the same ASI, is its preload register. The counter and preload register must be writable. Readability is an implementation parameter.

When enabled, the counter does one of the following:

— increments and generates an overflow signal when it passes maximum value

— decrements and generates an equal_zero signal when it reaches zero

— decrements and generates an underflow signal when it passes zero

The direction of counting and the form of strobe signal are implementation parameters. The strobe signal causes the contents of the associated preload register to be loaded into the counter and the counter continues counting.

If the strobe signal is delivered to an output pin, then it may be used to control periodic events in the external system such as DRAM refresh. If the strobe signal is delivered to an interrupt request line, then it may be used to periodically activate service routines such as polling external requests for service that do not activate vectored priority interrupts. Each counter is enabled by an associated bit in a system control register field when the bit is set to one. The counter is disabled by the associated bit being set to zero. The system control register is mapped into memory at ASI 1 or 0xC1. The address of the system control register and the mapping of counters to enable bits are implementation parameters.

5.5.3 Simple Timers

Timers ranging from 8 to 32 bits may be implemented to support more demanding timing tasks. The number of bits and the number of timers are implementation parameters.

Clocking of the timers and associated prescalers or reload registers is done with either or both a timer clock or an asynchronous external signal. Which one or both is an implementation parameter.

The timer clock is synchronized with the processor clock and may have the same period or may be divided down. The ratio of processor clock to timer clock is an implementation parameter. If divided down, transitions of the timer clock may be synchronized with the rising or falling edge of the processor clock. This is also an implementation parameter.

Asynchronous external signals are gated by the timer clock for internal synchronization. Therefore, the minimum duration of the asynchronous signal for its zero condition and for its one condition is some multiple of the timer clock period, which is an implementation parameter.

Each timer can be independently programmed to operate in one of the following five modes:

— Mode 0: Periodic Interrupt Mode

— Mode 1: Timeout Interrupt Mode

— Mode 2: Square Wave Generator Mode

— Mode 3: Software Trigger Watchdog Mode

— Mode 4: External Trigger Watchdog mode

Figure 10 below shows a block diagram of timers, prescalers and clock options. Prescalars and timer counters may be driven by the timer clock or an asynchronous external signal. Those timer units that have prescalers may also drive their timer counter with the prescaler output.

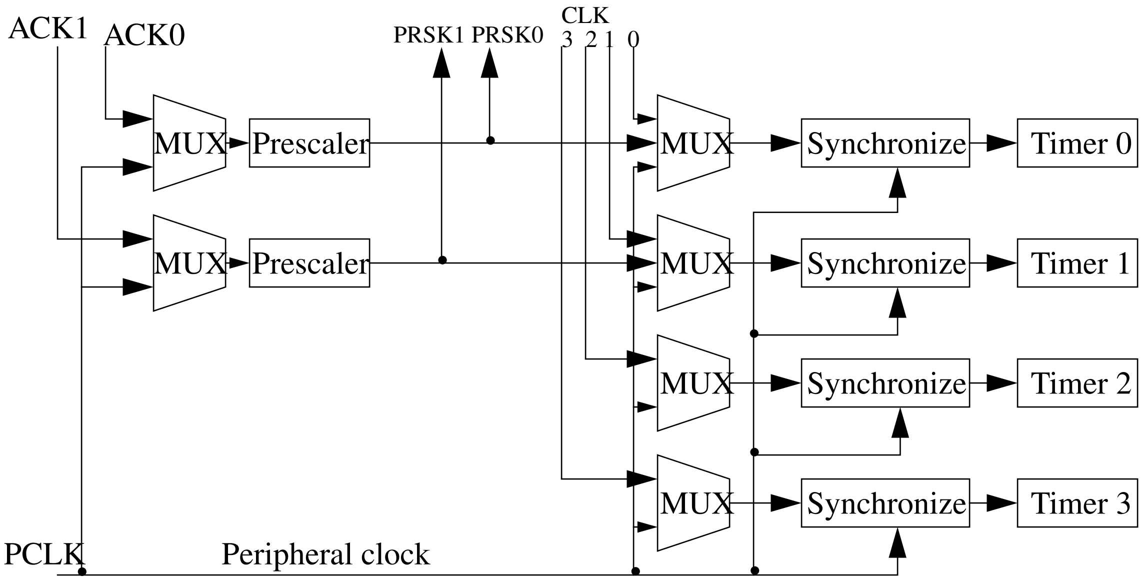

Figure 10: Timer Prescaler Block Diagram

Each timer unit has three or four internal registers that allow the program to control and monitor its operation. These registers are mapped into ASI 1 or 0xC1 at successive word addresses as shown in Table 7. Each starting address, TUC-REGn, is aligned on a quad word boundary (address modulo 16=0).

Address	Register	Required Access	Reset State
TUC-REGn + 0	Prescale control/reserved	Read/Write	1
TUC-REGn + 4	Timer control	Read/Write	0
TUC-REGn + 8	Reload value	Read/Write	0
TUC-REGn + C	Count value	Read	0

Table 7: Timer Unit Control Register Memory Map

Determination of timers having prescalers is an implementation parameter. Likewise, which count values can be written is an implementation parameter.

5.5.3.1 Prescaler Control Registers

Three fields are defined for each prescaler control register.

(1) Prescalar counter value- This determines the frequency of the prescaler counter output signal. The value of this field is written into the prescaler counter when this field is written and the timer clock is the prescaler counter clock or when prescaler timeout (counter reaches zero) occurs. This field must be greater than zero. A value of one produces the maximum prescaler counter output frequency, on half of its input frequency. The number of bits for this field is 8 to 16 and is an implementation parameter.

(2) Prescaler output select- This selects the prescaler output clock rate as the prescaler counter output frequency divided by the power of two indicated by this field. Zero means the prescaler output clock rate is the prescaler counter output frequency. One means the output clock rate is the prescaler counter output frequency divided by two. Two means the output clock rate is the prescaler counter output frequency divided by four, etc. Note that the maximum output clock rate is half the prescaler input frequency. The number of bits for this field is 2 to 14 and is an implementation parameter.

(3) Enable external clock- This one bit field enables asynchronous external signals when the prescaler input clock is one. When zero, the timer clock is the prescaler input clock.

All undefined bits are reserved and one bit is reserved for device test purposes.

5.5.3.2 Timer Control Registers

Eight fields are defined for each timer control register. A three bit Mode field selects which timer mode is active.

Mode Field	Operating Mode
0	Periodic Interrupt
1	Timeout Interrupt
2	Square Wave Generator
3	Software Trigger Watchdog
4	External Trigger Watchdog
5	Reserved
6	Reserved
7	Reserved

Table 8: Timer Control Register Entries

A three bit event field selects the condition for which the timer event input signal is active as an event gate or trigger depending on the mode.

Event Field	Active Gate/Trigger Event	Applicable Modes
0	Low Level Gate	0,1,2
1	High Level Gate	1.1.2
2	Rising Edge Trigger	4
3	Falling Edge Trigger	4
4	Rising/Falling Edge Triggers	4
5	Reserved	
6	Reserved	
7	Reserved	

Table 9: Timer Event Fields

A two bit Output Signal field selects the state of the output signals when the timer is stopped.

Output Signal Field	Timer Inactive Output State
0	Remains in current state
1	External clock
2	Prescaler output clock
3	Reserved

Table 10: Timer Output Signal Field

A one bit Invert field modifies the output signal. If Invert is one, then the actual output signal is the normal output signal inverted. If Invert is zero, then the actual output signal is the normal output signal.

A two bit Clock Select field selects the input clock to the timer counter.

Clock Select Field	Counter Clock Source
0	Timer clock
1	External Clock
2	Prescaler output clock
3	Reserved

Table 11: Timer Clock Select Field

A one bit Count Enable field, when one, enables the timer counter for counting. When Count Enable is zero, the counter stops. While enabled, the counter will not count until its specified input occurs. If a timer does not have a prescaler or if its prescaler output clock is not selected as its counter clock source, then configuring the timer counter and starting it can be done with a single STA instruction.

If a timer is to use its prescaler output clock as its counter clock source, then the prescaler configuration, the counter configuration, and starting can be done with a single STDA instruction since the two control registers are mapped into adjacent memory addresses that are double word aligned. Note that the first cycle of the counter action may differ from the later ones because the prescaler becomes active one store memory time before the timer counter. If the prescaler uses the timer clock to drive its counter, then the prescaler counter will be reloaded at the second memory cycle when the timer counter is loaded with its reload value.

A one bit Input Signal status field, which must be read only, allows the program to examine the Input Signal. A one bit Output Signal status field, which must be read only, allows the program to examine the Output Signal.

All undefined bits are reserved and one bit is reserved for device test purposes.

5.5.3.3 Prescaler Operation

Figure 11 shows an example prescaler block diagram consisting of an 8 bit counter, 7-divide by 2 flip-flops and selector logic

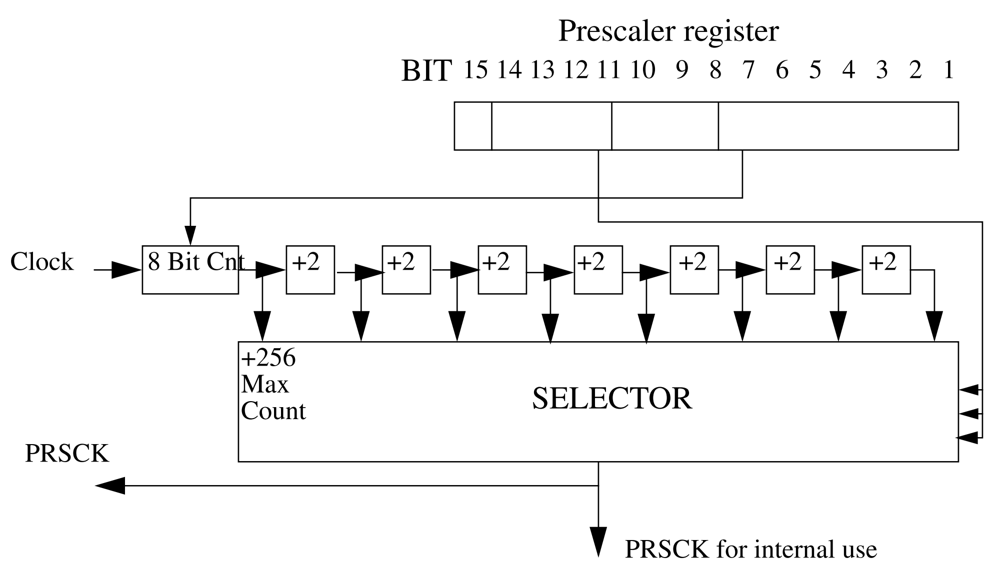

Figure 11: Prescaler Block Diagram

(1) Prescaler Counter:

The prescaler counter is loaded with the prescaler count value field concurrently with that field being written into the prescaler control register when the timer clock drives the counter. The counter then begins to decrement at its clock frequency and generates an output each time it reaches zero. This output is delivered to the first flip-flop input in the divide by 2 flip-flop cascade and the output selector. This output also reloads the counter with the prescaler count value field and continues the count.

If the associated timer counter uses the prescaler output clock as its input clock and if the prescaler uses the timer clock as its input, then when the timer reload value is loaded or reloaded into the timer counter, the prescaler count value is loaded into the prescaler counter and a fresh prescaler count cycle begins.

When the prescaler counter is driven by the external clock, the count value is loaded into the counter only when it reaches zero.If the count value is changed in the prescaler counter register, it will not be loaded into the counter until the counter reaches zero, finishing the previous count sequence. To reduce this delay, switch the prescaler counter drive to timer clock, then change the count value and switch to external clock.

(2) Prescaler Divide by 2 Cascade:

At each flip-flop output in the divide by 2 cascade, the frequency is halved. Each output is delivered to the next flip-flop input in the chain and the output selector.

The flip-flops in the divide by 2 cascade are cleared whenever the prescaler counter is loaded or reloaded.

(3) Prescaler Output Selector

The selector logic selects the counter output or one of the flip-flop outputs in the divide by 2 cascade as determined by the prescaler output select field of the prescaler control register. Code 0 selects the counter output. Code 1 selects the first divide by 2 flip-flop output. Code 2 selects the second divide by 2 flip-flop output, etc.

When one of the divide by 2 flip-flop outputs is the selected prescaler output clock then the duty cycle will be 50%. However, when the counter output is the selected prescaler output clock, then the output will be one more than zero except at the highest frequency.

The counter output is one until the counter decrements to one. Then the counter output is zero for one count cycle. Then when the counter reaches zero, the counter output returns to one while the counter is reloaded and begins a new count down. Therefore the prescaler output clock is zero for one count cycle and one all the rest of the count cycles.

However, if the prescaler count value is one, which selects the maximum prescaler counter frequency, the counter output is forced to zero at the same time the prescaler count value is loaded into the counter. The next cycle, with the counter at zero, the counter output goes to one and the counter is reloaded. For this case, the prescaler output clock has a 50% duty cycle.

5.5.3.4 Timer Operation and Timer Operating Modes

Figure 12 shows a block diagram of a timer.

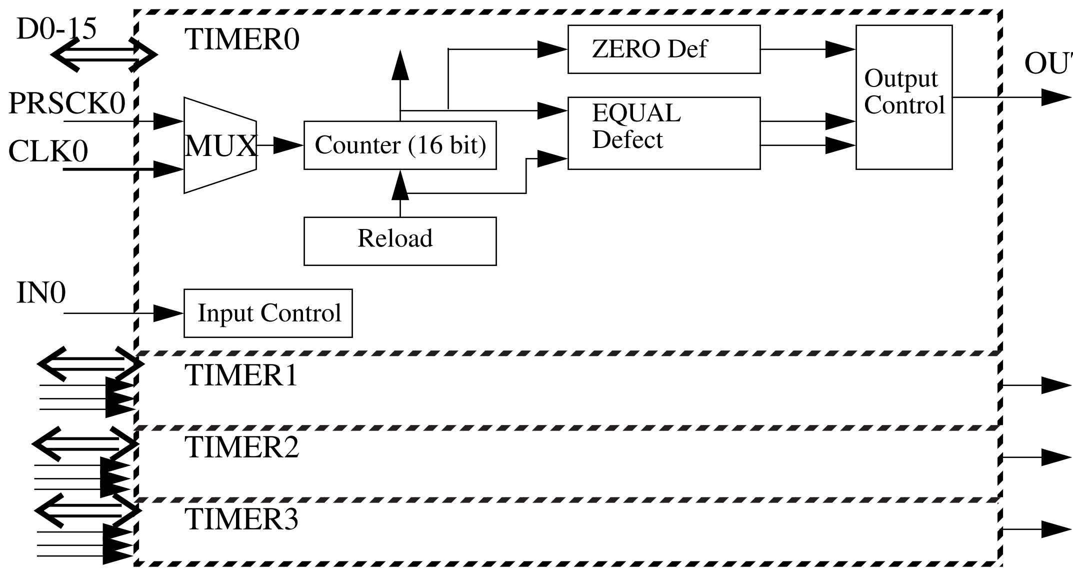

Figure 12: Timer Block Diagram

The data path to the processor is capable of both input and output. In addition to the timer clock, there are two other clock inputs, prescaler output clock and external clock. Also there is an In Signal that serves as input to count gating logic or event trigger logic. Finally, there is the timer Out Signal which may be delivered to an output pin and used to control external system activities or delivered to interrupt request lines and used to periodically activate service routines or used to activate exception routines if events fail to happen within preset time limits.

The In Signal can be used as a gating signal in Modes 0, 1, 2 and 3 to mask the timer counter input clock and temporarily stop the timer. It can be used in Mode 4 as a trigger event to start a new timer count sequence.

To use the In Signal as a gating signal in Modes 0, 1, 2 and 2, the Event field is set to make In active when one or when zero. When active, clocks to the input of the timer counter are inhibited and the counter does not count.

To use the In Signal as a triggering signal in Mode 4, the Event field is set to make the trigger event a rising edge, falling edge or both a rising and falling edge. When In Signal generates a trigger event, timer output is forced to value of Invert bit in timer control register, reload value is forced into timer counter and a new timer count sequence begins. At time-out, timer output changes to not Invert bit value.

The Out Signal is used to indicate timeout, occurrence of an event at the In Signal, or counter reaching half reload value during the count sequence. The Out Signal control determines the value of the Out Signal when the timer is stopped. The Invert bit determines the inactive level of Out Signal when the timer is running.

When the Invert bit is 0, for each mode, the following conditions set and reset the Out Signal:

Mode	Set Out Signal	Reset Out Signal
0	Timeout	Writing reload register/reading counter
1	Timeout	Writing reload register/reading counter
2	Timeout	Counter = half reload register
3	Timeout	Writing reload register
4	Timeout	Trigger event occurs at In Signal

Table 12: Output Signal Conditions

Set and Reset are swapped when the Invert bit is 1.

Timers are stopped following processor reset. Timer operation is initialized in all modes by first writing the timer mode into the Mode field of the Timer Control Register, setting the Count Enable field to 1 and writing any other appropriate fields of the Timer Control Register. Timer operation in modes 0, 1, 2 and 3 begins when the reload register is written. Then the reload value is set in the counter and decrementing begins.

Timer operation in mode 4 begins when a trigger event occurs at In Signal. Then the reload value is set in the counter and decrementing begins.

Once operating, each timer stops in the various modes as follows:

Mode	Stop Timer
0	Writing TCR, In Signal active
1	Writing TCR, In Signal active, timeout
2	Writing TCR, In Signal active
3	Writing TCR, In Signal active, timeout
4	Writing TCR, timeout

Table 13: Timer Stop Modes

Note that timers can be stopped in all modes by writing to the Timer Control Register.

With respect to timer operating modes, each timer can be independently programmed to operate in one of the following five modes:

Mode	Operation
0	Periodic Interrupt Mode
1	Timeout Interrupt Mode
2	Square Wave Generator Mode
3	Software Trigger Watchdog Mode
4	External Trigger Watchdog Mode

Table 14: Timer Operating Modes

The selection of a particular operating mode is controlled by the value of the Mode field in the Timer Control Register.

(1) Periodic Interrupt:

The Out Signal is initially set to the timer stopped state as determined by the Out Signal control of TCR, Timer is enabled, Count Enable =1 and Mode =0. When the reload register is written with the reload value, the reload value is set into the counter, the counter begins decrementing, and Out Signal is driven to the value of Invert.

When the counter reaches zero, timeout, the Out Signal changes to **NOT** Invert and remains at this level until the counter is read or reload register is written.However, the counter is automatically set with the reload value and continues decrementing. When the counter is read or reload register is written, Out Signal returns to Invert level.

(2) Time Out Interrupt:

This mode differs from Mode 0 at timeout. In Mode 1, the timer halts instead of reloading and decrementing the counter. Then, when the count register is read or the reload register is written, the Out Signal returns to Invert level, the counter is set with the reload value and begins decrementing again,

(3) Square Wave Generator:

This mode differs from Mode 0 in the transition of Out Signal.

The Out Signal is initially set to the timer stopped state as determined by the Out Signal control of TCR, Timer is enabled, Count Enable =1 and Mode =2. When the reload register is written with the reload value, the reload value is set into the counter, the counter begins decrementing, however, Out Signal remains at the initial value.

When the counter decrements to half of reload value, Out Signal is driven to Invert value. When counter reaches 0, timeout, Out Signal changes to **NOT** Invert value. The counter is set with the reload value and continues decrementing. After the first count sequence, Out Signal will be approximately a square wave.

For reload value =1, Out Signal will be at Invert and not Invert level one timer count cycle each, with a period of two timer count cycles.

For reload value =N, N>1, Out Signal will be at Invert level for Int(N/2) timer count cycles. Out Signal will be at **NOT** Invert level for Int((N+1)/2) +1 timer count cycles. The period will be N+1 timer count cycles.

(4) Software Trigger Watchdog:

The Out Signal is initially set to the timer stopped state as determined by the Out Signal control of TCR, Timer is enabled, Count Enable =1 and Mode = 3. When the reload register is written with the reload value, the reload value is set into the counter, the counter begins decrementing, and Out Signal is driven to the value of Invert.

When counter = 0, timeout, Out Signal changes to not Invert value and remains at this value. The timer halts. However, writing to the reload register before timeout, updates the counter with the reload value and delays timeout.

After the timer halts, it can be restarted by writing to the reload register. The reload value is set into counter and the watchdog count restarts.

(5) Hardware Trigger Watchdog:

The Out Signal is initially set to the timer stopped state as determined by the Out Signal control of TCR, Timer is enabled, Count Enable =1 and Mode =4. Then the reload register is written with the reload value.

When a trigger event occurs at In Signal, the reload value is set into the counter, the counter begins decrementing, and Out Signal is driven to the value of Invert.

When counter = 0, timeout, Out Signal changes to **NOT** Invert value and remains at this value and the timer halts. However, occurrences of another trigger event at In Signal before timeout, updates the counter with the reload value and delays timeout.

The timer is restarted after halting at timeout by another trigger event at In Signal. The In Signal trigger event is determined by the Event field in TCR and can be a rising edge, falling edge or both.

6 Diagnostic F acilities

6.1 Introduction

As an option, a V8E SPARC can be equipped with a Debug Support Unit (DSU). This unit provides functions such as

- setting hardware breakpoints on instruction and data, on address and value;
- single stepping;
- instruction trace generation;
- reading and writing of on chip registers;
- emulation.

They are detailed in the subsequent sections of this chapter.

Two implementations with different emphasis are defined. Both support the above features. They are:

- a trace enhancing implementation
- a pin effective implementation.

6.1.1 The trace enhancing implementation.

This implementation makes use of a number of extra pins to allow a sufficient number of instruction address bits per SPARC instruction to reconstruct a full address trace. The same pins are used to control the further diagnostic features introduced above and to be detailed in the subsequent sections of this chapter. Implementor defined further features may be added to this implementation; these further features are not subject of the V8E specification.

6.1.2 The pin effective implementation.

This implementation relies an a JTAG interface for control as well as for information transport to and from the DSU and for control as well as for information transport to and from any further features that can be reached via the DSU.

This implementation also relies heavily on DMA; using JTAG and DMA it allows software independent access to all DSU functions described in the subsequent sections. Since the JTAG, in this implementation, is a DMA master and the DSU is a DMA slave, it can not only reach the DSU functions specified here and any implementor defined further DSU functions but also: - any (further) DMA mapped sources and destinations, on or off chip; - any (further) ASI mapped sources and destinations, on or off chip; - any (further) memory mapped sources and destinations, on or off chip, e.g. . on chip or off chip RAM; . on chip or off chip ROM. Obviously, all of the above, being mapped on space that can be reached via software, can also be read, written and controlled via software.

Unlike conventional debuggers, this implementation can be used before system I/O is available to e.g. load RAM; to access internal address and data buses and to break-point or single-step through code sequences.

6.2 List of features

The following features will be detailed in subsequent detailed sections in a next release of this spec.

— BREAK:

- HARDW ARE BREAK

  INSTRUCTION ADDRESS MATCH BREAK

  DATA ADDRESS AND DATA MATCH BREAK

  DSU Register Write Exception Break

  DSU Register Read/Write Exception Break

  External (-EMU_BRK Pin) Break

  Hardware Break Request

- Softw are Break

  Hardware/Software Break Request Disable

  Ret-Break (Return from Break)

  Disable_match_match flag

  Trap_Disabled_Break_Point

  Break on RETT

— Single Step:

- Single Step general beha vior
- Single Step beha vior over trap
- Single Step beha vior on RETT

— ICE Port:

— Debug Mode and State:

— Instruction Trace

— DSU registers:
Obviously, the DSU register sets of the two implementations are not identical.

Annex A Programming Techniques

(Informative)

A.1 Overview

This section provides information to assist in programming the SPARC-V8E embedded processor. It covers the additional instructions, Divide Step and Scan, provided by SPARC-V8E. Code fragments are provided to illustrate the use of these instructions. This section presumes familiarity with the SPARC-V8 Programmer's Model and SPARC assembly language as specified in The SPARC Architecture Manual, Version 8.

A.2 Division Performance Using DIVScc

A.2.1 divs1 - divide signed, 1 word dividend

Signed division of 32 bit dividend by 32 bit divisor produces a signed 32 bit quotient and a signed 32 bit remainder (same sign as dividend or zero if exact). Since the only overflow is divide by zero, this routine does not check for divide by zero, leaving it up to caller to test and abort just after the call. Division without fault takes 47 to 58 cycles assuming each instruction takes one cycle, except `retl` which takes two.

```
!DIVISION SUBROUTINE - DIVS1
!This subroutine for signed division of 32 bit dividend by 32 bit divisor
!produces 32 bit signed quotient and 32 bit remainder using divide
!step instruction. Remainder is zero if division is exact
!or same sign as original dividend if not. There is no check for divide
!by zero. It is not possible to overflow with non zero divisor. If the
!calling routine knows that divide by zero cannot happen, no test is
!needed. If divide by zero is possible, a simple test just after the call
!can abort the division. Division without fault takes 47 to 58 cycles.
!Exact division with last partial remainder =0 takes 47 cycles. Exact
!division with last partial remainder = +/-divisor, as happens with
!non-restoring division algorithms, takes 51 or 52 cycles.Inexact
!division, with non-zero final remainder, takes 54 to 58 cycles.

!call so:
!    mov %l1,%o0!dvdnd->o0
!    orcc %g0,%l2,%o2!dvsr->o2 & test
!    call divs1!DIVISION SUBROUTINE CALL
```

```
!    be dvby0 !abort division if divide by zero
!
! Register Map
!reg#
!out0 dividend/remainder
!out1 quotient
!out2 divisor
!out4 scratch for final remainder calculations
!out5 absolute value of divisor
!y   initially sign extension of dividend/successive partial remainders
!call to divs1 must be made with cc indicating sign of divisor
!
.global divs1
divs1:mov %g0,%y!0 -> Y
     mov %o2,%o5!copy divisor in o5, D
     bl,a 1f
     sub %g0,%o5,%o5!if divsr neg, D=-divsr
1:   tst %o0  !initialize cc for first divide step
              !with sign dividend for signed divide
     bl,a 2f
     mov -1,%y!-1 -> Y only if dvdnd neg
2:   divscc %o0,%o5,%o1!divide step 1
!leave original dividend in o0
!do partial remainders & quotient in o1
!don't change cc except by divscc until last divide step done
     divscc %o1,%o5,%o1!divide step 2
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
```

```
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1!divide step 32
     be 6f    !if final remainder is zero,
              !go fix quotient polarity
     mov %y, %o4 !final remainder from Y to o4
     bg 4f    !skip ahead if rmdr+; continue if rmdr-
     addcc %o4,%o5,%g0 !is neg rmdr + abs divsr =0
     mov %g0,%o4!clear rmdr. if not, don't clear
     tst %o0  !test original dvdnd
     bl 5f    !if neg, go check neg Q
     tst %o1  !sign Q
     ba 5f
     add %o4,%o5,%o4!if orig dvdnd pos and final rmdr neg,
              !correct rmdr; then go check neg Q
4:   subcc %o4,%o5,%g0 !is pos rmdr - abs divsr =0
     be,a 6f  !if so, go fix quotient polarity and
     mov %g0,%o4!clear rmdr. if not, don't clear
     tst %o0  !test original dvdnd
     bge 5f   !if pos, go check neg Q
     tst %o1  !sign Q
     sub %o4,%o5,%o4!if orig dvdnd neg and final rmdr pos,
              !correct rmdr; then go check neg Q
5:   bl,a 6f  !skip ahead if Q pos
     add %o1,1,%o1!if neg Q, 1's complement to
              !2's complement; annul if pos Q
6:   tst %o2  !check original divisor sign
     bl,a 7f
     sub %g0,%o1,%o1!if neg divsr, negate quotient
7:   retl     !exit
     mov %o4,%o0!with correct remainder in o0
```

A.2.2 divs2 - divide signed, 2 word dividend

Signed division of a 64 bit dividend by a 32 bit divisor produces a signed 32 bit quotient and a signed 32 bit remainder with the same sign as dividend or zero if exact. Division with divide by zero fault takes 6 cycles. Division with non zero divisor overflow fault takes 17 to 23 cycles. Division without fault takes 49 to 60 cycles. This assumes each instruction takes one cycle except `retl` which takes two.

```
!DIVISION SUBROUTINE - DIVS2 REVA
!This subroutine for signed division of 64 bit dividend by 32 bit divisor
!produces 32 bit signed quotient and 32 bit remainder using divide
!step instruction. Special treatment is given to borderline overflow
!with absolute value quotient = 2^31 to support math operator INTEGER
!PART OF: Q=-2^31 does not overflow; Q=+2^31 overflows as before but
!with different overflow code. Remainder is zero if division is exact
!or same sign as original dividend if not. There is a check for divide
!by zero and a check for overflow with non-zero divisor. Check for divide
!by zero is kept separate to support possible SUN recommended trap for
!divide by zero. In applications where user knows numerical ranges or
!controls them, these checks can be omitted. Division with divide by zero
!fault takes 6 cycles; sets overflow flag in condition code; leaves
```

```
!0xfffff800 in register out3. Division with non-zero divisor overflow
!takes 17 to 23 cycles (17 or 19 if original dividend plus, 18 or 23 if
!original dividend minus); sets overflow flag in condition code; leaves
!0x800 in register out3. Division leading to absolute value quotient =
!2^31 takes 20 cycles if original dividend plus, 23 cycles if original
!dividend minus. It leaves correct remainder in register out0, -2^31 in
!out1 as quotient and 0 in out3. It clears overflow cc if actual quotient
!is -2^31 and sets overflow cc if actual quotient is +2^31. Division
!without fault takes 49 to 60 cycles; clears overflow flag in condition
!code; leaves 0 in register out3. Exact division with last partial
!remainder =0 takes 49 cycles. Exact division with last partial
!remainder=+/-divisor, as happens with non-restoring division
!algorithms, takes 53 or 54 cycles. Inexact division, with non-zero final
!remainder, takes 56 to 60 cycles.
!call so:
!    mov %l0,%o0 !msh dvdnd->o0
!    mov %l1,%o1 !lsh dvdnd->o1
!    call divs2 !DIVISION SUBROUTINE CALL
!    orcc %g0,%l2,%o2 !dvsr->o2 & test
!
!Register Map
!reg#
!out0    msh dividend/remainder
!out1    lsh dividend/quotient
!out2    divisor
!out3    overflow indication
!overflowdivide by zero/0xfffff800 and V=1
!overflowdivide by non-zero/0x800 and V=1
!overflowquotient =+2^31/0 and V=1
!no overflow/0 and V=0
!out4    scratch for final remainder calculations
!out5    absolute value of divisor
!y       msh dividend/successive partial remainders
!call to divs2 must be made with cc indicating sign of divisor
!
.global divs2
divs2:bne 0f  !go on if divisor not zero
    mov %o2,%o5!copy divisor in o5, D
    sethi 0x1fffff,%o3!divide by zero indicator
    retl     !exit with
    addcc %o3,%o3,%o3 !overflow set
0:  bl,a 1f
    sub %g0,%o5,%o5!if divsr neg, D=-divsr
1:  mov %o0,%y!msh dvdnd->Y
    tst %o0  !initialize cc for first divide step
             !with sign dividend for signed divide
    bl 2f    !skip ahead for negative dividend
    divscc %o1,%o5,%o1!divide step 1
!don't change cc except by divscc until last divide step done
    bl 3f    !ok if different
    mov %g0,%o3!clear overflow indicator
    srl %o1,1,%o4!get lsh rmdr
    bg 8f    !if msh rmdr >0 then overflow
    subcc %o4,%o5,%g0!if lsh rmdr <D then Q is +/-2^31
```

```
     bge 8f   !& o4 is correct final rmdr
              !check if overflow on Q = +2^31
     sethi 0x200000,%o1!set -2^31 -> Q
              !else overflow
     tst %o2  !if original divisor >0
     bg,a 9f  !which implies quotient =+2^31
     addcc %o1,%o1,%g0!set ovrlfw cc with o3 = 0
9:   retl     !exit
     mov %o4,%o0!with correct remainder in o0
8:   sethi 0x200001,%o3!overflow divide by non-zero indicator
     retl     !exit with
     addcc %o3,%o3,%o3 !overflow set
2:   bge 3f   !ok if different
     mov %g0,%o3!clear overflow indicator
     mov %y,%o0!get msh rmdr
     addcc %o0,1,%g0!is it -1
     bne 8f   !if <-1 then overflow
     srl %o1,1,%o4!get lsh rmdr except for leading 1
     sethi 0x200000,%o1!set -2^31 ->Q
     or %o1,%o4,%o4!insert leading 1 in lsh rmdr
     addcc %o4,%o5,%g0!if lsh rmdr >-D then q is +/-2^31
     ble 8f   !& o4 is correct final rmdr
              !check if overflow on Q = +2^31
              !else overflow
     tst %o2  !if original divisor <0
     bl,a 9f  !which implies quotient =+2^31
     addcc %o1,%o1,%g0!set ovrlfw cc with o3 = 0
9:   retl     !exit
     mov %o4,%o0!with correct remainder in o0
8:   sethi 0x200001,%o3 !overflow divide by non-zero indicator
     retl     !exit with
     addcc %o3,%o3,%o3!overflow set
3:   divscc %o1,%o5,%o1!divide step 2
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
     divscc %o1,%o5,%o1
```

```
    divscc %o1,%o5,%o1
    divscc %o1,%o5,%o1
    divscc %o1,%o5,%o1
    divscc %o1,%o5,%o1
    divscc %o1,%o5,%o1
    divscc %o1,%o5,%o1
    divscc %o1,%o5,%o1
    divscc %o1,%o5,%o1
    divscc %o1,%o5,%o1!divide step 32
    be 6f    !if final remainder is zero,
             !go fix quotient polarity
    mov %y, %o4 !final remainder from Y to o4
    bg 4f    !skip ahead if rmdr+; continue if rmdr-
    addcc %o4,%o5,%g0!is neg rmdr + abs divsr =0
    be,a 6f  !if so, go fix quotient polarity and
    mov %g0,%o4!clear rmdr. if not, don't clear
    tst %o0  !test original dvdnd
    bl 5f    !if neg, go check neg Q
    tst %o1  !sign Q
    ba 5f
    add %o4,%o5,%o4!if orig dvdnd pos and final rmdr neg,
             !correct rmdr; then go check neg Q
4:  subcc %o4,%o5,%g0!is pos rmdr - abs divsr =0
    be,a 6f  !if so, go fix quotient polarity and
    mov %g0,%o4!clear rmdr. if not, don't clear
    tst %o0  !test original dvdnd
    bge 5f   !if pos, go check neg Q
    tst %o1  !sign Q
    sub %o4,%o5,%o4!if orig dvdnd neg and final rmdr pos,
             !correct rmdr; then go check neg Q
5:  bl,a 6f  !skip ahead if Q pos
    add %o1,1,%o1!if neg Q, 1's complement to
             !2's complement; annul if pos Q
6:  tst %o2  !check original divisor sign
    bl,a 7f
    sub %g0,%o1,%o1!if neg divsr, negate quotient
7:  retl     !exit
    mov %o4,%o0!with correct remainder in o0
```

A.2.3 divu1 - divide unsigned, 1 word dividend

Unsigned division of a 32 bit dividend by a 32 bit divisor produces an unsigned 32 bit quotient and an unsigned 32 bit remainder that is positive or zero if exact. Since only overflow is divide by zero, this routine does not check for divide by zero, leaving it up to caller to test and abort just after the call. Division without fault takes 39 cycles. If remainder is of no interest and only the quotient corresponding to INTEGER(dvdnd/dvsr) or FLOOR(dvdnd/dvsr) for unsigned numbers is wanted then the last steps of this routine can be modified as indicated and quotient only unsigned division will take 36 cycles. This assumes each instruction takes one cycle except `retl` which takes two.

```
!DIVISION SUBROUTINE - DIVU1
!This subroutine for unsigned division of 32 bit dividend by 32 bit
!divisor produces 32 bit unsigned quotient and 32 bit remainder using
!divide step instruction. Remainder is zero if division is exact
!or positive if not. There is no check for divide by zero. It is not
!possible to overflow with non zero divisor. If the calling routine
!knows that divide by zero cannot happen, no test is needed. If divide
!by zero is possible, a simple test just after the call can abort the
!division. If not aborted, division takes 39 cycles; clears overflow
!flag; leaves 0 in register out3.
!If remainder is of no interest and only the quotient corresponding to
!INTEGER(dvdnd/dvsr) or FLOOR(dvdnd/dvsr) for unsigned numbers is wanted
!then the last steps of this routine can be modified as indicated and
!quotient only unsigned division will take 36 cycles.
!call so:
!    mov %l1,%o1!dvdnd->o1
!    orcc %g0,%l2,%o2!dvsr->o2 & test
!    call divu1!DIVISION SUBROUTINE CALL
!    be dvby0 !abort division if divide by zero
!
!Register Map
!reg#
!out0 remainder
!out1 dividend/quotient
!out2 divisor
!out3 0 if divide by non zero
!y   initially zero/successive partial remainders
!
.global divu1
divu1:mov %g0,%y!0->Y
    orcc %g0,0,%o3!initialize cc for first divide step
              !with positive sign for unsigned divide
              !clear divide by zero indicator
    divscc %o1,%o2,%o1!divide step 1
!don't change cc except by divscc until last divide step done
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
```

```
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    ! retl   !exit for quotient only divide
    divscc %o1,%o2,%o1!divide step 32
    !ALL the following steps may be omitted for quotient only divide
    bl 1f    !skip ahead if rmdr-
    mov %y,%o0!final rmdr from Y to o0
    retl     !exit
    addcc %o0,0,%o0!clear ovrflw cc if on
1:  retl     !exit
    addcc %o0,%o2,%o0!correct rmdr & clear ovrflw cc if on
```

A.2.4 divu2 - divide unsigned, 2 word dividend

Unsigned division of a 64 bit dividend by a 32 bit divisor produces an unsigned 32 bit quotient and an unsigned 32 bit remainder that is positive or zero if exact. Division with divide by zero fault takes 6 cycles. Division with non zero divisor overflow fault takes 9 cycles. Division without fault takes 42 cycles. This assumes each instruction takes one cycle except `retl` which takes two.

```
!DIVISION SUBROUTINE - DIVU2
!This subroutine for unsigned division of 64 bit dividend by 32 bit
!divisor produces 32 bit unsigned quotient and 32 bit remainder using
!divide step instruction. Remainder is zero if division is exact
!or positive if not. There is a check for divide by zero and a check for
!overflow with non-zero divisor. Check for divide by zero is kept
!separate to support possible SUN recommended trap for divide by zero. In
!applications where user knows numerical ranges or controls them, these
!checks can be omitted. Division with divide by zero fault takes 6
!cycles; sets overflow flag in condition code; leaves 0xfffff800 in
!register out3. Division with non-zero divisor overflow takes 9 cycles;
!sets overflow flag in condition code; leaves 0x800 in register out3.
!Division without fault takes 42 cycles; clears overflow flag in
!condition code; leaves 0 in register out3.
!call so:
!   mov %l0,%o0!msh dvdnd->o0
!   mov %l1,%o1!lsh dvdnd->o1
!   call divu2!DIVISION SUBROUTINE CALL
!   orcc %g0,%l2,%o2!dvsr->o2 & test
!
!Register Map
!reg#
```

```
!out0    msh dividend/remainder
!out1    lsh dividend/quotient
!out2    divisor
!out3    overflow indication
!overflow divide by zero/0xfffff800 and V=1
!overflow divide by non-zero/0x800 and V=1
!no overflow/0 and V=0
!y       msh dividend/successive partial remainders
!call to divs2 must be made with cc indicating if divisor zero
!
.global divu2
divu2:bne 1f  !go on if divisor not zero
     mov %o0,%y!msh dvdnd->Y
     sethi 0x1fffff,%o3!divide by zero indicator
     retl     !exit with
     addcc %o3,%o3,%o3 !overflow set
1:   subcc %o0,%o2,%g0!is msh dvdnd < dvsr
     bcs 2f   !ok if so
     orcc %g0,0,%o3  !initialize cc for first divide step
              !with positive sign for unsigned divide
              !clear overflow indicator
     sethi 0x200001,%o3!overflow divide by non-zero indicator
     retl     !exit with
     addcc %o3,%o3,%o3 !overflow set
2:   divscc %o1,%o2,%o1!divide step 1
!don't change cc except by divscc until last divide step done
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
     divscc %o1,%o2,%o1
```

```
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1
    divscc %o1,%o2,%o1!divide step 32
    bl 3f    !skip ahead if rmdr-
    mov %y,%o0!final remdr from Y to o0
    retl     !exit
    addcc %o0,0,%o0!clear ovrflw cc if on
3:  retl     !exit
    addcc %o0,%o2,%o0!correct rmdr & clear ovrflw cc if on
```

A.3 SCAN Instruction Examples

A.3.1 Software floating point with SCAN

The following code fragment shows post normalization of floating point add or subtract for the case where the result requires calculating the difference of the magnitudes of the numbers. The IEEE754 format, which is used in SPARC-V8 architecture is assumed. This uses sign, offset exponent, hidden leading bit when normalized and fraction. Only the logic of normalize numbers is shown here. Number values are in sign and magnitude form rather than two's complement.

```
bit  31| 30 23| 22 0normalized values
fields |  e|  f 0<e<255
             x = (-1)^s * 2^(e-127) * (1 + f*2^-23)
```

The operation is x+y=z or x−y=z. If subtract, then sign y is complemented. The magnitudes of the numbers have to be compared and the one with the lesser exponent right shifted to align its decimal point with the greater exponent. If exponents are equal, magnitudes must be compared if signs differ to determine what the sign of the result will be. This is assumed to have taken place before the code fragment shown here, which shows the logic of handling numbers with different signs and different exponents. Symbol x points to the larger number; y to smaller.

```
sethi 0x3fe, %g5!mask for sign and exponent with and
        !or for fraction with andn
sll %g5,1,%g4
xor %g4,%g5,%g4!single one at bit 23 for hidden bit
srl x,23,%g2
and %g2,0xff,%g2!x exponent
srl y,23,%g3
and %g3,0xff,%g3!y exponent
sub %g2,%g3,%g1!alignment difference
andn y,%g5,%g3!y fraction
or %g3,%g4,%g3!y hidden bit
srl %g3,%g1,%g2!downshift y magnitude to g2
sub %g0,%g1,%g1!complement of shift
sll %g3,%g1,%g3!upshift left over y for test
```

```
     addcc %g3,%g3,%g0!test left over for rounding
              !note: not IEEE754 rounding here
     andn x,%g5,%g1!x fraction
     or %g1,%g4,%g1!x hidden bit
     subx %g1,%g2,%g1!difference of magnitudes with
              !simple rounding
     scan %g1,0,%g2!scan difference for leading one.
              !Use of 0 as the scan mask is because
              !of sign magnitude arithmetic assumed
              !in this example. Leading 8 bits are
              !guaranteed to be zero because of
              !format. Question is, how many more
              !till the first one?
              !If two's complement arithmetic had
              !been assumed, then there could have
              !been leading ones or leading zeros
              !depending on sign of result. Then
              !instead of 0 as mask, scan would have
              !used %g1 as mask as well as value.
              !Question would have been, how many
              !leading bits are the same as the sign?
     subcc %g2,32,%g0!test if all significant bits lost
     blu 1f   !use unsigned compare for future compatibility
     sub %g2,8,%g2!remove effect of format's 8 leading 0's
     !underflow due to loss of significant bits code would follow here

1:   sll %g1,%g2,%g1!normalize result
     andn %g1,%g4,%g1!hide leading bit
     srl x,23,%g3
     and %g3,0xff,%g4!x exponent in g4
     subcc %g4,%g2,%g0!test exponent underflow
     bgu 2f   !use unsigned compare for future compatability
     sub %g3,%g2,%g3!subtract normalization shift from
     !result sign and exponent
     !exponent underflow code would follow here

2:   sll %g3,23,%g3!place sign and exponent result in
              !format position
     retl     !exit(2 cycles)
     or %g1,%g3,z!combine with fraction
```

Each instruction in this code fragment runs one cycle out of the instruction cache except for the leaf routine which takes two. That's 32 cycles for this fragment. Without scan as a hardware instruction, the function would have to be performed as a software routine that takes 43 to 52 cycles for the usual cases. The fragment would take 74 to 83 cycles, more than double the cycles. A software substitute for scan would consume instruction cache space. Attempts to speed up the binary tree search in the software routine by look up tables based on leading bits would consume data cache space.

A.3.2 Run length encoding with SCAN

The following code fragment shows compression of long binary strings by looking for runs of all ones or all zeros and coding these so that lossless reconstruction is possible. For the example, runs less than four in length are ignored and directly transmitted and runs greater than sixteen are broken up for coding efficiency and coding simplification. Best compression occurs for low information content long binary strings such as background sections of black and white raster lines.

```
code value
00000reserved
00001"
00010"
00011"
------------------------------
0010000001... or 11110...
00101000001... or 111110...
001100000001... or 1111110...
...
011110000 0000 0000 0001... or 1111 1111 1111 1110...
100000000 0000 0000 0000 1... or 1111 1111 1111 1111 0...
-------------------------------------------------------
100010001...
100100010...
100110011...
...
111101110...
-----------------
11111toggle
```

The code fragment omits starting up the loop, reloading buffers with new data, storing code and terminating the loop. Symbol x points to data segment in some register ready for compression and symbol y points to its immediate successor. Symbol z points to some register that will hold code for compression data.

```
0:   scan x,x,%g1!scan for how many bits are same as msb.
             !g1 = 1 to 31 or >32 if all in x register.
             !x is used as both the value to be scanned(rs1)
             !and the mask(rs2).
     subcc %g1,4,%g0!test if run at least length 4
     bgeu 1f  !use unsigned compare for future compatability
     subcc %g1,16,%g0!test if run greater than length 16
             !handle fixed length code, g1<4
     srl x,28,%g2!extract leading 4 bits of x as compression code
     or %g2,16,%g2!insert leading bit of code for fixed length
     sll x,3,x!shift rest of x in 2 steps
     addcc x,x,x!complete x shift and test last of 4 bits
     bcs 2f   !separate cases for 1 or 0
     addcc x,x,%g0!test without shifting first of remaining bits
     bcs 3f   !if last out bit =0 and first remaining bit =1
     mov 1,%g4!set new low priority toggle indicator
     ba 3f
     mov 0,%g4!otherwise clear toggle indicator
             !fixed length code overwrites any pending toggle
```

```
2:   bcc 3f   !if last out bit =1 and first remaining bit =0
     mov 1,%g4!set new low priority toggle indicator
     mov 0,%g4!otherwise clear toggle indicator
              !fixed length code overwrites any pending toggle
3:   srl y,28,%g3!extract leading 4 bits of y
     or x,%g3,x!move them to right end of x
     sll y,4,y!shift rest of y with incoming trailing zeros
     ba 5f
     subcc %g5,4,%g5!decrement counter of how many bits of x left
              !handle run length code
1:   blu 4f   !skip ahead if run less than 16
              !use unsigned compare for future compatability
     sll %g4,1,%g4!shift incomming toggle indicator to higher
              !priority; handle runs at least 16
     mov 16,%g2!set compression code to 16
     sll x,16,x!ignore leading 16 bits of x and shift rest of x
     srl y,16,%g3!extract leading 16 bits of y
     or x,%g3,x!move them to right end of x
     sll y,16,y!shift rest of y with incomming trailing zeros
     ba 5f
     subcc %g5,16,%g5!decrement counter of how many bits of x left
              !handle runs of length 4 to 15
4:   mov %g1,%g2!set compression code to scan result
     sub %g0,%g1,%g1!complement scan result
     sll x,%g2,x!ignore leading g2 bits of x and shift rest of x
     srl y,%g1,%g3!extract leading 32-g1 bits of y
     or x,%g3,x!move them to right end of x
     sll y,%g2,y!shift rest of y with incomming trailing zeros
     subcc %g5,%g2,%g5!decrement counter of how many bits of x left
     or %g4,1,%g4!toggle following compression code too
              !one compression code to go
5:   bgu 6f   !skip ahead if there are still bits of x left
              !use unsigned compare for future compatability
     subcc %g6,1,%g6!decrement counter of code fields left
!code for reloading y and shifting part of it into x if the old y had
!trailing zeros and resetting g5 to 32-#trailing zeros.
 ...
6:   bg 7f    !skip ahead if room for more codes
     andcc %g4,2,%g0!test if toggle has priority
              !code for storing codes and reinitializing g6
 ...
7:   sll z,5,z!make room for new code
     be,a 0b  !if g4 bit1 off then no additional code
              !if g4 bit1 on then insert toggle code first
     or z,%g2,z!insert new data code
     andn %g4,2,%g4!clear high priority toggle indicator
!without disturbing low priority toggle indicator
     ba 5b!check on how much code space left and append toggle
     or z,0x1f,z!back through 5,6,7 just once
 ...
```

Each instruction in this code fragment runs one cycle out of the instruction cache if it is in the active path for a particular case. Scan is in the active path for all cases. Without hardware implementation of Scan, the function would require a software subroutine taking 43 to 52 cycles instead of only 1. Additionally, that routine would consume instruction cache space. Alternate versions that might attempt to speed up the binary tree search with table look up using leading bits as an index would consume data cache space.

Annex B Alternative Window Usage Models

(Informative)

B.1 Overview

This section provides an alternative to the standard SPARC programming model for the SPARC-V8E processor. SPARC-V8 processors provide a large number of general purpose registers. At any instant, the SPARC CPU has 32 working registers available. They are divided into 8 global registers, and 24 registers that compose a current overlapped “register window”. The SPARC Architecture Manual, Version 8 requires that the number of register windows on any implementation fall between 2 and 32.

The abundance of registers and the window register model provided by SPARC-V8 incurs several drawbacks and performance penalties when implemented in high performance, real-time applications associated with embedded systems such as SPARC-V8E. One such problem encountered is the large number of registers that may need to be saved at context switch time leading to high context switch overhead. Another, is the difficulty in predicting the number of registers that will need to be saved at context switch time. This results in all registers being saved with associated performance penalties.

In embedded applications, the following factors take precedence:

- Reduced average context-switch times
- Constant (or small worst-case deterministic) context-switch and procedure-call times

There are several alternatives known to accommodate the above criteria.

Note:

See Section D.8 of the SPARC Architecture Manual, Version 8, for descriptions of other register-window usage models. (Note that model “[C]” in D.8 is the one described in more detail below.)

B.2 Single Register Window Model

An alternate mechanism for the SPARC-V8E window register programming model, The Single Register Window Model, is described. This Model:

— Avoids the typical window "overflow" processing overhead

— Reduces the number of registers that need to be saved on a context switch

This model avoids using the standard SPARC-V8 register windowing mechanism; instead, it treats the SPARC processor as a conventional CPU with a flat set of 32 general purpose registers. The compiler generates code to save registers around procedure calls when necessary. Dedicating a register window to a single process is possible. If a process has a window dedicated to it, the context of a process is always available without reference to memory. Thus, little memory access is required for a context switch. In the example illustrated in Figure 3, four of the eight register windows are dedicated to four processes. When an interrupt or another context switching event is detected, the SPARC processor automatically switches to a new window (which is not shown in diagram). Thus, the local registers between the reserved windows are reserved for interrupt handling.

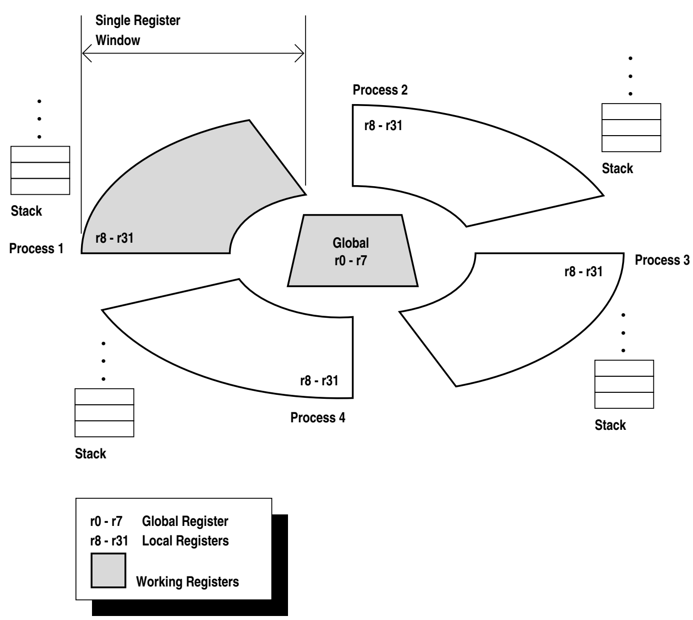

Figure 13: Alternative Window Model for Machine with 8 Register Windows

Table3 below shows how a Single Register Window Model compiler will use registers. Notice that the "in," "local," and "out" grouping have been replaced by a flat register file %r0 through %r31. Even with two registers reserved for future use, 11 registers are now available for local variables, thereby minimizing expensive load/store operations.

Register	Use	Comments
r31	Stack Pointer	also referred to as **sp**
r30	Frame Pointer	also referred to as **fp**
r25 - r29	Scratch Registers	used by the compiler for temporary values
r24	Return value	start of quad precision value or address of **struct**
r16 - r23	Input Parameters	additional parameters placed on the stack
r15	Return Address	address of the procedure call instruction
r4 - r14	Register Variables	local variables
r3	Special Use	*reserved* for the user
r1 - r2	Reserved	*reserved* for future (position-independent code)
r0	Zero Value	always contains the value zero

Table 15: Register Usage in the Single Register Window Model

B.3 Conclusion

The above alternative window register model, The Single Window Register Model, provides enhanced functionality for the SPARC-V8E processor. Both improvements in the speed of procedure calls and returns and the guaranteeing of constant or worst-case deterministic context switch times are accommodated by the enhanced window register model. Other solutions have been proposed and may be equally efficient.

Compatibility Note:

Assembly language or computer output from SPARC-V8 ABI conforming programs will not run on the single register window model.

Annex C Summary of Operation Codes, ASIs and ASRs

(Normative)

C.1 Operation Codes

— Divide Step-DIVScc: op=2, op3= 011101_2 /($1D_{16}$)

— Scan-SCAN: op=2, op3=101100_2/ ($2C_{16}$)

Compatibility Note

The SCAN operation code, op3= $2C_{16}$, conflicts with the SPARC-V9 opcode for MOVcc.

C.2 ASI Assignments

The following revision is made to the recommended ASI assignments for ASI's 30_{16}-FF_{16} from Table I-1 in the SPARC-V8 Architecture Manual:

ASI	Function
30-6F	unassigned
70-7F	*reserved* for diagnostic facility
80-BF	*reserved**
C0-EF	unassigned*
F0-FF	*reserved* for diagnostic facility*
	*may be accessible in user mode

Table 16: ASI Assignments

C.3 ASRs

— ASR-17: Trap SVT Flag, bit 0

Annex D List of options

(Normative)

D.1 Introduction

The list included in this annex shows which V8E extensions to the SPARC-V8 architecture there are, in which combinations they are allowed or advised, and where in the SPARC-V8E spec they are described.

The options are numbered according to the chapters in the SPARC-V8E spec in which they are described; the sub numbering does, though, not necessarily follow the section numbering and subsection numbering as in the SPARC-V8E spec..

D.2 Instructions

The following options are defined.
They can be implemented not at all, separately or in any combination:

2.1. Divide Step:	See Chapter 2.1
2.2. Scan	See Chapter 2.2
2.3. Multiply Accumulate	See Chapter 2.3
2.4. Alternate Window Pointer	See Chapter 2.4
2.5. Partial WRPSR	See Chapter 2.5
2.6. Non Privileged ASI Access	See Chapter 2.6

D.3 MMU

None or just one of the options below can be implemented:

3.1. Basic reference MMU	See SPARC-V8 spec
3.2. Embedded Reference MMU	See Chapter 3.2
3.3. Cacheability Control	See Chapter 3.3

D.4 Traps

The option below may or may not be supported:

4.1. Single Vector Trapping See Chapter 4.

D.5 Peripheral Extensions

5.1. Input Handler: See Chapter 5.2
Any number of input handlers can be supported. They will usually but not necessarily be used as inputs to other options as described in chapter 5.

5.2. Interrupt Controller: See Chapter 5.3.1 and 5.3.2.
One or more such controllers may be implemented. They will usually be preceded by Input Handlers <5.1>, and if more than one Interrupt Controller <5.2> is implemented, then their outputs are combined as an Extended Interrupt Controller <5.3>.

5.3. Extended Interrupt Controller: See Chapter 5.3.3.
If more than one Interrupt Controller <5.2> is implemented then they should be combined into an Extended Interrupt Controller <5.3>.

5.4. Integrated Interrupt Request Controller: See Chapter 5.4.
None or one “IIRC” <5.4> can be implemented.
<5.4> can not be implemented together with circuitry <5.2> or <5.3>.

5.5. Programmable Pulse Generators: See Chapter 5.5.1.
Any number of such Counter/Timer/Pulsers may be implemented. They will usually be preceded by Input Handlers <5.1>; their outputs will usually be connected to Interrupt Controllers <5.2> but they can also be otherwise connected; that is implementor defined.

5.6. Simple Counters See Chapter 5.5.2.
Any number of Simple Counters may be implemented. They may be preceded by Input Handlers <5.1>; their outputs may be connected to Interrupt circuitry <5.2> or <5.4>

5.7. Simple Timers See Chapter 5.5.3.
Any number of Simple Timers may be implemented. They may be preceded by Input Handlers <5.1>; their outputs may be connected to Interrupt circuitry <5.2> or <5.4>.

D.6 Diagnostics

None or just one of the options below may be implemented: They share functionality; their implementation is optimized for different desires.

6.1. Trace enhancing DSU: See Chapter 6.1.1.
A next version of the SPARC-V8E spec will contain more details.

6.2. Pin effective DSU: See Chapter 6.1.2.
A next version of the SPARC-V8E spec will contain more details.