STMicroelectronics

ST240 core and instruction set architecture

Reference manual

8059133 Rev C

27 June 2008

BLANK

Reference manual

ST240 core and instruction set architecture

Introduction

The ST240 is a member of the ST200 family of cores.

This ST200 family of embedded processors use a scalable technology that allows variation in VLIW (very long instruction word) operation issue width, the number and capabilities of functional units and register files, and the instruction set.

The ST200 family includes the following features:

- parallel execution units, including multiple integer ALUs and multipliers
- architectural support for data prefetch
- predicated execution through select operations
- efficient branch architecture with multiple condition registers
- encoding of immediate operands up to 32-bits
- support for user/supervisor modes
- memory protection

The ST240 adds the following enhancements to this list:

- predicated execution through conditional loads and stores
- 64-bit loads and stores using pairs of 32-bit registers
- multiple floating point units
- saturating arithmetic and SIMD included in the multiple integer ALUs
- an integer divide and remainder unit
- instruction set support for unified L2 cache

Contents

Preface

ST200 document identification and control

Each book in the ST200 documentation suite carries a unique ADCS identifier of the form:

ADCS *nnnnnnnx*

where *nnnnnnn* is the document number, and *x* is the revision.

Whenever making comments on an ST200 document, the complete identification ADCS *nnnnnnnx* should be quoted.

ST200 documentation suite

The ST200 documentation suite comprises the volumes listed below.

ST240 Core and Instruction Set Architecture

This manual describes the architecture and the instruction set of the ST240 core as used by STMicroelectronics.

ST200 Micro Toolset User Manual

ADCS 8063762. This manual describes the ST200 Micro Toolset and provides an introduction to OS21. It covers the various cross tools and libraries that are provided in the toolset, the target platform libraries, how to boot OS21 applications from ROM. Information is also given on how to build the open source packages that provide the compiler tools, base run-time libraries and debug tools and how to set up an ST Micro Connect.

ST200 Micro Toolset Compiler Manual

(ADCS 7508723) This manual provides a detailed guide to using the ANSI C and C++ compiler drivers for compiling and linking source code to produce an executable binary. The compiler drivers are introduced in terms of how they fit into the complete ST200 toolchain. The manual then concentrates on the facilities provided by the compiler drivers to produce efficient code. It covers: command line options, predefined macros, supported pragmas, compiler optimization techniques, GNU C and C++ language extensions and `asm` construct, the assembly language and intrinsic functions.

ST200 Run-time Architecture Manual

ADCS 7521848. This manual describes the common software conventions for the ST200 processor run-time architecture.

OS21 User Manual

ADCS 7358306. This manual describes the royalty free, light weight, OS21 multitasking operating system.

OS21 for ST200 User Manual

ADCS 7410372. This manual describes the use of OS21 on the ST200 platforms. It describes how specific ST200 facilities are exploited by the OS21 API. It also describes the OS21 board support packages for ST200 platforms.

ST200 ELF Specification

ADCS 7932400. This document describes the use of the ELF file format for the ST200 processor. It provides information needed to create and interpret ELF files and is specific to the ST200 processor.

Conventions used in this guide

General notation

The notation in this document uses the following conventions:

- `sample code`, `keyboard input` and `file names`
- *variables* and *`code variables`*
- *`code comments`*
- **screens**, **windows** and **dialog boxes**
- **instructions**

Hardware notation

The following conventions are used for hardware notation:

- REGISTER NAMES and FIELD NAMES
- PIN NAMES and SIGNAL NAMES

Software notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF), briefly:

- Terminal strings of the language, that is, strings not built up by rules of the language, are printed in teletype font. For example, `void`
- Nonterminal strings of the language, that is, strings built up by rules of the language, are printed in italic teletype font. For example, *`name`*
- If a nonterminal string of the language starts with a nonitalicized part, it is equivalent to the same nonterminal string without that nonitalicized part. For example, `vspace-`*`name`*
- Each phrase definition is built up using a double colon and an equals sign to separate the two sides (‘`::=`’)
- Alternatives are separated by vertical bars (‘`|`’)
- Optional sequences are enclosed in square brackets (‘`[`’ and ‘`]`’)
- Items which may be repeated appear in braces (‘`{`’ and ‘`}`’)

Acknowledgements

The ST240 core is based on technology jointly developed by Hewlett-Packard Laboratories and STMicroelectronics.

1 Overview

This chapter provides an overview of the ST240 prcessor and to this reference manual.

1.1 VLIW overview

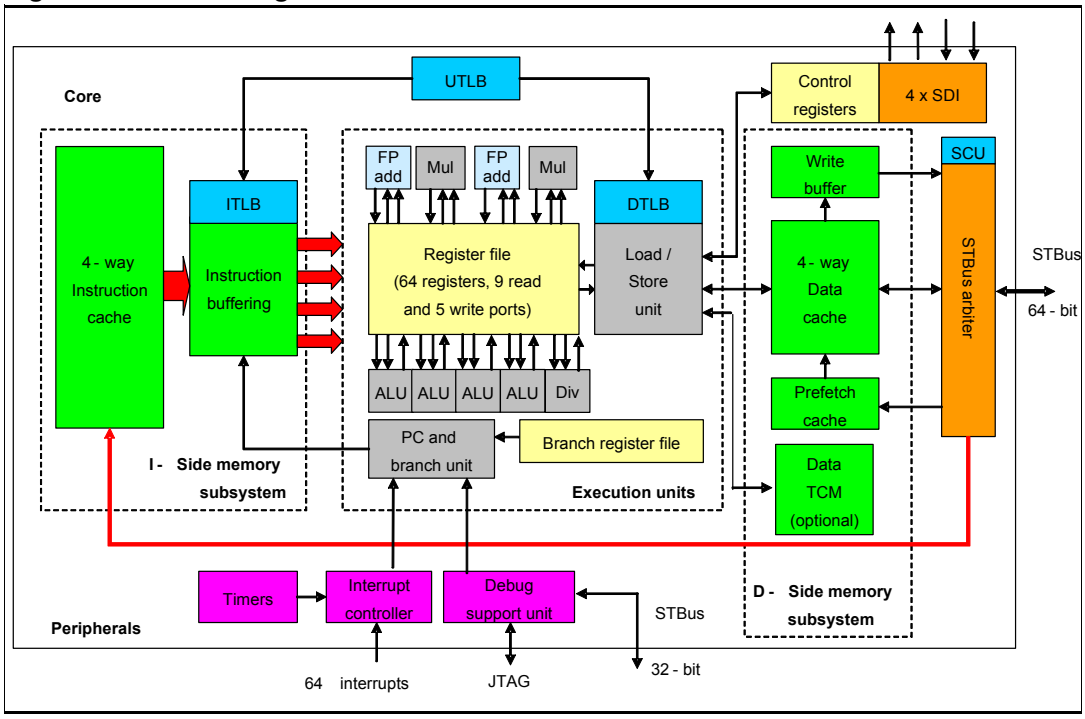
VLIW (very long instruction word) processors use a technique whereby more than one operation is performed in parallel within the same clock cycle. This is done in order to take maximum advantage of all the processor's available resources. The hardware implementation of a VLIW processor is significantly simpler than a corresponding multiple issue superscalar CPU because of the simplicity of the grouping and scheduling hardware; the complexity is passed to the instruction scheduling software (compiler and assembler) which is responsible for scheduling the parallel operations for maximum efficiency.

In the ST240, simple, RISC-like operations (known as "syllables") are grouped together into bundles so that a single bundle fills an instruction word. The operations in a bundle are then processed simultaneously. In most cases the operations complete simultaneously; however some results can be bypassed to subsequent operations prior to completion. This is discussed in *Chapter 6: Execution pipeline and latencies on page 33*.

1.2 ST240 overview

The ST240 includes the ST240 core and an associated peripherals block. *Figure 1* shows the arrangement of these components in a block diagram.

Figure 1. Block diagram of the ST240



1.3 Document overview

This manual describes the architecture and instruction set of the ST240. References are made to one specific implementation of the ST240. This section gives an outline of the document.

The ST240 is made up of the functional units described in *Chapter 2: Execution units on page 18*; these operate on data stored in the register files described in *Chapter 4: Architectural state on page 27*. The functional units are pipelined and behave as described in *Chapter 6: Execution pipeline and latencies on page 33*.

The ST240 provides standard user and supervisor modes.These modes are described in *Chapter 3: Operating modes on page 23*.

The ST240 has a defined set of rules that govern how individual syllables can be bundled together into a single instruction word. These rules are decribed in *Chapter 5: Bundling rules on page 31*.

ST240 arithmetic operations are described in *Chapter 7: Arithmetic operations on page 36*, logical operations in *Chapter 8: Logical operations on page 59* and single instruction multiple data (SIMD) instructions in *Chapter 9: SIMD operations on page 64*.

The handling of exceptions and interrupts is discussed in *Chapter 10: Traps (exceptions and interrupts) on page 82*.

The ST240 accesses memory through the memory subsystem described in *Chapter 12: Memory subsystem on page 110*. The memory subsystem provides protection and address translation by means of a Translation Lookaside Buffer; this is discussed in *Chapter 11: Memory translation and protection on page 93*.

ST240 support for multi-processing (MP) and multi-threading (MT) is described in *Chapter 13: Multi-processor and multi-threading support on page 132*.

The ST240 has four SDI ports for rapid communication with other devices and to avoid cache pollution when processing large amounts of data; these are described in *Chapter 14: Streaming data interfaces on page 140*.

Using the memory mapped control registers to control the state of the ST240 is described in the relevant chapters. The addresses of the control registers and PSW are listed in *Chapter 15: Control registers on page 145*.

The ST240 also provides a performance monitoring system to help with software optimization and debugging; this is described in *Chapter 21: Performance monitoring on page 192*.

Details of the following peripherals are also provided:

- timers in *Chapter 17: Timers on page 155*
- interrupt controller in *Chapter 19: Interrupt controller on page 163*
- the Debug Support Unit (DSU) in *Chapter 20: Debugging support on page 169*

The peripheral register addresses are listed in *Chapter 18: Peripheral addresses on page 157*.

The execution model is described in *Chapter 22: Execution model on page 199*. The execution of bundles is discussed in *Section 22.1: Bundle fetch, decode, and execute on page 199*, including the behavior of the machine when exceptions or interrupts are encountered.

Chapter 24: Instruction set on page 222 provides details of every operation, including instruction set encoding, syntax and semantics. The encoding of bundles is defined in *Section 24.1: Bundle encoding on page 222*.

The behavior of operations is specified using the notational language defined at the beginning of *Chapter 23: Specification notation on page 202* and continuing to *Section 23.3: Statements on page 209*. The descriptions identify where architectural state is updated and the latency of the operations.

A simple model of memory and control registers is described in *Section 23.5.2: Memory model on page 215* and *Section 23.5.3: Control register model on page 218* describes the techniques required when specifying load and store operations.

A Glossary of terms is provided in*: Glossary on page 496*.

2 Execution units

The ST240 includes a number of execution units working on two register files. The architecture permits one or more execution units with variable latencies. This chapter refers to the specific implementation of the architecture as shown in *Figure 1: Block diagram of the ST240 on page 15*, although other implementations may exist. In this implementation, there are:

- four arithmetic and logic units (ALU) (all of which can perform single instruction, multiple data (SIMD) operations)
- two multiply units (which can also perform SIMD, floating point and multiply-add operations)
- two floating point units which perform floating point addition, subtraction and conversions
- one divide and remainder unit
- one load/store unit
- one branch unit

A bundle may therefore contain up to four ALU operations, up to two multiply operations, up to two floating point operations, only one (or no) load/store operation and only one (or no) branch operation. The combinations of operations which may be executed in a single bundle are defined in *Chapter 5: Bundling rules on page 31*.

The two register files; the branch registers and the general purpose registers are described in *Chapter 4: Architectural state on page 27*. All other units are described in this chapter.

2.1 Arithmetic and logic units

The current implementation of the ST240 has four identical arithmetic and logic units (ALU). These also perform SIMD operations. Most of the supported operations produce results after one cycle; this means that an operation in the next bundle can immediately use the result as an operand.

Each operation accepts between one and three operands in the form of zero, one or two 32-bit values and zero, one or two 4-bit conditional values. The ALU then executes the appropriate operation and produces one or two results, depending upon the operation being carried out. The results can either be zero or one 32-bit value and zero or one 4-bit conditional value.

The integer operations supported are described in *Chapter 7: Arithmetic operations on page 36*, *Chapter 8: Logical operations on page 59* and *Chapter 9: SIMD operations on page 64*.

2.2 Floating point units

The ST240 has two identical floating point units. Each unit is pipelined with a pipeline depth of three cycles.

Each floating point operation accepts one or two operands in the form of 32-bit values and produces one 32-bit result.

For full details of the floating point operations see *Section 7.8: Floating point operations on page 51*.

2.3 Multiplication units

The ST240 has two identical multiplication units. Each unit is pipelined with a depth of three cycles and executes an operation every cycle. The multiplication units support integer, SIMD and floating point arithmetic.

Each multiplication unit accepts two 32-bit operands and produces a single 32-bit result. The multiplication operations supported are described in *Section 7.2: Multiplication operations on page 40*.

2.4 Divide and remainder unit

The ST240 has a single divide and remainder unit. It is pipelined with a depth of three cycles and executes an operation every cycle.

The divide and remainder unit accepts two 32-bit operands and produces a single 32-bit result.

Note: The divide and remainder unit will stall the pipeline if the result is not available within three cycles.

The supported divide and remainder operations are described in *Section 7.10: Divide and remainder operations on page 57*. The causes of the differing number of cycles taken for specific divide and remainder cases is explained there.

2.5 Load/store unit

The ST240 has a single load/store unit (LSU). The load/store unit is pipelined with a depth of three cycles, and executes an operation every cycle.

The load/store unit can take up to one 4-bit operand, two 32-bit operands and one 64-bit operand and may produce no result, a 32-bit or 64-bit result depending on the operation. The load/store operations supported are described in the *Section 12.5: D-side memory subsystem on page 118*.

Memory access protection and translation is implemented by the translation lookaside buffer (TLB), which is part of the memory subsystem. The TLB also controls the cache behavior of data accesses, see *Chapter 11: Memory translation and protection on page 93*.

Uncached accesses or accesses that miss the data cache both cause the load/store unit to stall the pipeline in order to ensure correct operation.

2.5.1 Memory access

The ST240 uses a single 32-bit address space to address the external memory system. Peripherals and control registers are mapped within this address space.

All cacheable memory transactions are made via the data cache. The data cache determines if an external memory access (using the STBus) is required to complete the request.

Note: The data cache does not allocate a line when servicing a write miss.

Uncached accesses are performed directly on the memory system via the STBus as described in *Uncached load and stores on page 120*.

2.5.2 Addressing modes

Every load/store operation supports one of the following addressing modes:

- the multi-processor/multi-threading support operations (**ldwl**, **stwl**) support register addressing only
- all other load/store operations support immediate plus register

2.5.3 Alignment

All load/store operations operate on data stored on the natural alignment of the data type; that is:

- long words on long word boundaries
- words on word boundaries
- half-word on half word boundaries

A long word load or store uses a pair of contiguous registers to contain the data. More details are given in *Data widths on page 119*.

2.5.4 Control registers

A portion of the address space is dedicated to control registers; these are described in *Chapter 15: Control registers on page 145*. Load and store operations that access the control register space are not translated by the TLB and do not access the data cache.

2.5.5 Cache coherency control

The ST240 provides operations which enable software control of cache coherency. These operations include:

- invalidate operations that remove data from a cache without updating other parts of the memory system
- flush operations that copy data from a cache into other parts of the memory system
- purge operations that copy data from a cache into other parts of the memory system and remove the data from the cache

The ST240's operations allow:

- invalidating individual lines of the data cache
- purging individual lines of the data or instruction cache[a]
- purging entire sets from the data or instruction cache

The ST240 also allows some operations to be applied only to the first level of cache. This permits the first level instruction and data caches to be made coherent efficiently.

a. When applied only to the instruction cache there is no distinction between a purge and an invalidation as the data is always clean. The distinction is made when a unified L2 cache is in use as the instruction cache purge operations will purge, not invalidate, entries from the L2 cache.

2.5.6 Conditional load, stores and prefetches

The ST240 uses conditional loads, stores and prefetches to support if-conversion and pipelining of loops. These operations execute only if a test condition proves true. In the case of the condition being false, they execute as **nop** operations with no side-effects.

2.5.7 Data-side tightly coupled memory

The ST240 may include a data-side tightly coupled memory (DTCM), which is a memory mapped section of RAM contained within the LSU. The LSU accesses the DTCM in parallel with the data cache, and the DTCM may also be accessed from the STBus target port.

Please refer to your datasheet to see if a DTCM is included. Also see *Section 12.5.6: D-side tightly coupled memory on page 125*.

2.5.8 Level 2 cache

The ST240 supports a unified level 2 cache. This cache behaves as an extension of the level 1 data and instruction caches. Details are given in *Section 12.5.8: Level 2 cache support on page 126*.

2.5.9 Multi-processor and hardware multi-threading support

The ST240 instruction set includes operations that enable the ST240 to be part of a multi-processor cache-coherent system, and to support hardware multi-threading. These operations are described in *Chapter 13: Multi-processor and multi-threading support on page 132*.

2.6 Branch unit

The ST240 has one branch unit. This unit supports branches, gotos, calls and returns as listed in *Table 1*. These operations reference any branch register: see *Section 4.3: Branch register file on page 28*, but only one general purpose register: see *Section 4.2.1: Link register on page 27*.

The relevant operations are listed in *Table 1*. The operands that appear in this table are as follows:

- **B_{BCOND}** is the branch register used for the condition code for a conditional branch
- **BTARG** is the PC relative immediate offset of the branch, **goto**, call or return
- **$r63** is the link register

Table 1. Branch unit operations

operation	description
br B_{BCOND}, BTARG	PC relative conditional branch: take branch if condition is *TRUE*.
brf B_{BCOND}, BTARG	PC relative conditional branch: take branch if condition is *FALSE*.
goto BTARG	PC relative unconditional goto.
goto $r63	Go to link register.
call $r63	Call link register, stores return address in link register.
call $r63 = BTARG	PC relative call, stores return address in link register.
return $r63	Return to link register. Equivalent to **goto $r63**, but used to allow call/return prediction in future implementations.
rfi	Return from interrupt. This operation is not simply a branch; it updates the architectural state and empties the pipeline.

All successfully-taken branches, gotos, calls and returns incur a penalty of one cycle of stall. The **rfi** operation causes five cycles of stall as the pipeline is emptied.

3 Operating modes

The ST240 provides standard user and supervisor modes. These modes enable a *user* program to run securely under the control of a *supervisor* program. The ST240 maintains security by:

- using the TLB to control the user program's access to specific memory locations
- limiting the operations that the user process can execute
- restricting the user program's access to some ST240 resources

The user program cannot undermine the security of memory accesses because a user program cannot re-program the TLB. Neither can the user program change itself into a supervisor program that can re-program the TLB.

The ST240 also provides resources to enable a supervisor program to debug a user program.

In addition to these standard modes, the ST240 provides a *debug* mode. Debug mode permits an external debugger to control the operation of the core and to debug programs running on the core. The ST240 provides some specific debugging resources; these are only accessible in debug mode. In debug mode, the debugging system is protected from code running in supervisor or user mode.

The ST240 can be in two different states when in debug mode; user debug and supervisor debug. The difference between these two states is significant when accessing memory. In summary, the machine has three *operating modes* (user mode, supervisor mode, debug mode) and four *operating states* (user state, supervisor state, user debug state, supervisor debug state).

Note: The "state" terminology is used only when it is necessary to distinguish between the two different states corresponding to debug mode.

Refer to *Chapter 10: Traps (exceptions and interrupts) on page 82*, and *Chapter 20: Debugging support on page 169* for more information.

3.1 Representation of each mode

The current operating mode is represented by two bits in the PROGRAM STATUS WORD (PSW) (see *Section 4.4: Program status word on page 28*), DEBUG_MODE and USER_MODE, as shown in *Table 2*.

Table 2. Representation of the operating modes

DEBUG_MODE	USER_MODE	Operating state	Operating mode
0	0	Supervisor	Supervisor
0	1	User	User
1	0	Debug (Supervisor)	Debug
1	1	Debug (User)	

3.2 Access to resources

The following sections describe resources that have restricted access in the various operating modes.

3.2.1 Control registers

All control registers can be accessed (that is, read from or written to) in debug mode. Control registers that are specifically related to a host that is debugging the ST240 by means of the debug support unit (DSU) are only accessable in debug mode. See *Section 20.3: Debug support unit on page 177* for more details. All other control registers can be accessed in supervisor mode.

Control registers relating to OS functions (such as TLB registers and trap handler registers) cannot be accessed in user mode. Some control registers (such as the streaming data interface (SDI) registers) have programmable permissions. To allow access in user mode, appropriate permissions must be programmed in supervisor or debug mode.

For the complete list of control registers and access permissions see *Chapter 15: Control registers on page 145*.

3.2.2 PSW updating operations

The operations **rfi** and **pswmask** update the PSW. These operations can be executed only in supervisor or debug mode. An attempt to execute them in user mode causes an illegal instruction trap to be raised. See *Section 20.2.3: Software breakpoints*.

3.2.3 TLB pages

TLB entries can be programmed in order to restrict access to the memory that they map in user mode, supervisor mode, or both (see *Table 43: TLB_INDEX bit fields on page 98*). The USER_MODE PSW bit determines access to the restricted pages. As described in *Section 3.3: Transitioning between modes and states on page 25*, entering debug mode clears the USER_MODE PSW bit and therefore forbids access to pages without supervisor mode permission.

It is necessary to set the USER_MODE PSW bit to access pages with user mode only permissions when the ST240 is in debug (supervisor) mode this causes a transition from the debug (supervisor) state to the debug (user) state, as shown in *Figure 2*.

3.2.4 Peripherals

The TLB also controls access to peripherals. The DSR_PERMISSIONS control register provides a second level of protection. The access is programmable and can be one of the following:

- access in any mode
- access in supervisor or debug mode only
- access in debug mode only

Other STBus initiators are permitted access to the DSRs. There are further details in *Section 20.3.2: Shared register bank on page 178*.

3.3 Transitioning between modes and states

Figure 2 and *Figure 3* show the possible transitions between the user, supervisor, debug (user) and debug (supervisor) states.

Figure 2. State transitions due to PSW updating operations

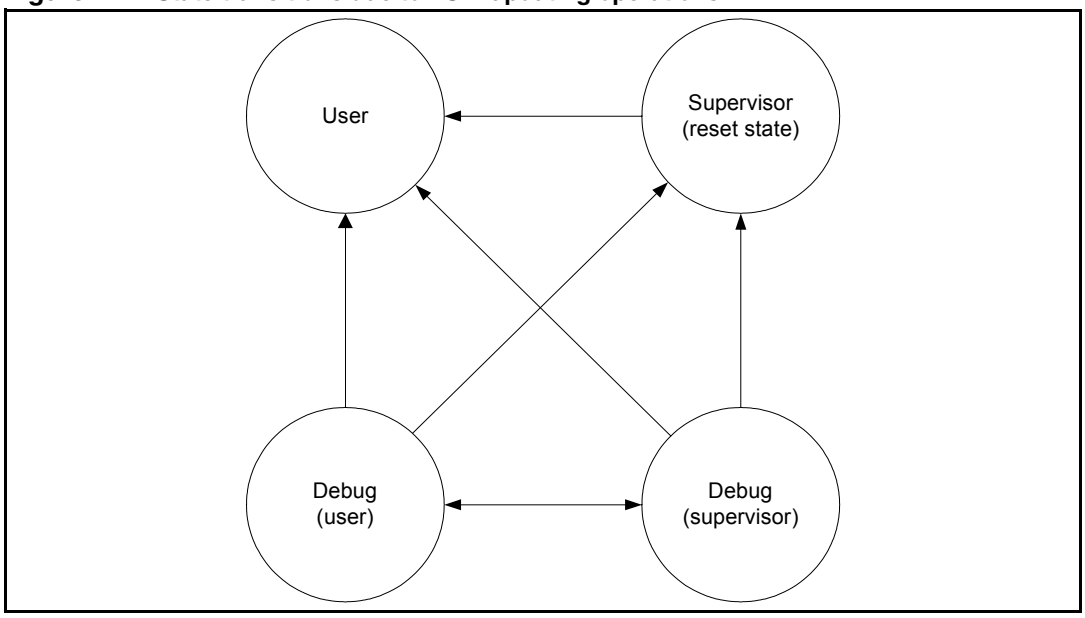
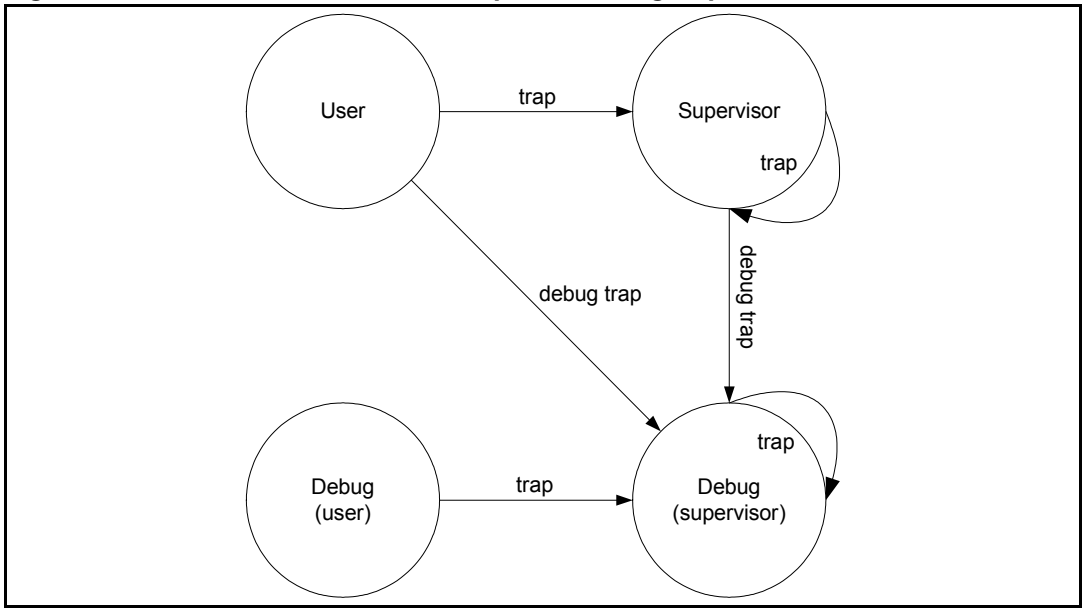


Figure 3. State transitions due to traps and debug traps



3.3.1 Supervisor mode to user mode

On reset, the ST240 starts in supervisor mode. After boot, an OS such as Linux switches into user mode before starting to run the first user application. This transition is made by executing a PSW updating operation.

3.3.2 User to supervisor mode

The transition from user to supervisor mode is made by means of a trap. The transition may occur for various reasons. Some of these reasons are listed here.

- Interrupt - the user application is being interrupted.
- TLB fault - the user code has failed TLB checks and requires assistance from the OS.
- Breakpoint - the user code is being debugged.
- Syscall - the user code required a function that needs OS support to complete (for example, `printf()`).
- Other trap - any other trap is likely to be fatal for user code. An example of this is an STBus error.

3.3.3 Debug mode entry and exit

The ST240 can enter debug mode only by means of a debug trap. There are several sources of debug trap listed in *Chapter 20: Debugging support on page 169*. The ST240 exits from debug mode by means of an operation to update the PSW.

Note that a normal trap in debug mode causes the ST240 to remain in debug mode. Details are given in *Trap handler with the default debug handler on page 184*.

No debug traps are accepted when the ST240 is already in debug mode. If a debug interrupt is requested, it remains pending until the ST240 exits debug mode and causes the ST240 to re-enter debug mode immediately. Debug exceptions are not raised in debug mode. See *Section 20.2: Core debugging support on page 171*.

Attempts to enter debug mode from supervisor mode with a PSW updating operation fail silently (the PSW bit is not updated and no exception is raised).

4 Architectural state

This chapter describes the architectural state of the ST240 core, which consists of the following elements:

- program counter
- register file
- branch register file
- program status word
- control registers

4.1 Program counter

The program counter (PC) contains a 32-bit byte address pointing to the beginning of the current bundle in memory. The two LSBs of the PC are always zero as the PC is always word aligned.

The PC value is not directly available in a register. It is made visible by three methods.

- By execution of an **addpc** operation, which adds an immediate value to the current PC.
- By taking a trap which saves the current PC into SAVED_PC.
- A delayed version of the PC is available from the debug support virtual PC register (DSVPC), see *Section 20.3.4: Debug support virtual PC register on page 180*.

4.2 Register file

The general purpose register file contains 64 words of 32 bits each. These are named R0 to R63.

Reading register zero (R0) always returns the value zero. Writing values to R0 has no effect on the architectural state.

64-bit load/store operations access contiguous pairs of registers. For details see *Long word accesses on page 119*.

4.2.1 Link register

Register R63, the architectural link register, is used by the **call** and **return** mechanism. R63 is updated by explicit register writes and the **call** operation. The link register may be used as a general register although some restrictions apply to accessing it, see *Section 6.5: Link register restrictions on page 34*.

4.3 Branch register file

The branch register file contains eight 4-bit branch registers, B0 to B7. The functions of these registers are as follows.

- They can represent boolean conditions for scalar operations that read branch bits: that is, conditional branches, loads, stores and the **addcg** operation.
- They can hold 4-bit condition codes for SIMD operations.

The ALU performs a boolean conversion from an arbitrary 4-bit value to 0x0 or 0x1 on a branch register that is specified as an input operand to any scalar operation.

SIMD operations read and write 4-bit values. 16-bit SIMD comparisons set bit 0 with the result of the lower comparison and bit 2 with the result of the higher comparison. 8-bit SIMD comparisons perform four comparisons - one for each bit of the result.

All Scalar operations write 0x1 for true and 0x0 for false.

The ST240 uses the **mov** operation to move 4-bit values between branch registers and general purpose registers. When transferring from a general purpose register to a branch register, the ST240 transfers the lower four bits of the general purpose register only. When transferring from a branch register to a general purpose register, the core zero-extends the 4-bit value from the branch register to 32- bits.

The macro **convib** performs a boolean conversion on a general purpose register by writing the result to a branch register and the macro **convbi** performs the reverse boolean conversion on a branch register by writing the result to a general purpose register.

4.4 Program status word

The program status word (PSW) contains control information that affects the operation of the ST240.

4.4.1 Bit fields

The PSW contains the bit fields listed in *Table 3*.

Table 3. PSW bit fields

Name	Bit(s)	Writable	Reset	Comment
USER_MODE	0[a]	RW	0x0	1: the core is in user mode 0: the core is in supervisor mode
INT_ENABLE	1	RW	0x0	1: internal and external interrupts are enabled 0: internal and external interrupts are disabled
TLB_ENABLE	2	RW	0x0	1: address translation is enabled 0: address translation is disabled
Reserved	3	RO	0x0	Reserved (This was TLB_DYNAMIC on the ST231.)
Reserved	4	RO	0x0	Reserved (This was SPECLOAD_MALIGN_EN on the ST231.)

Table 3. PSW bit fields (Continued)

Name	Bit(s)	Writable	Reset	Comment
DTCM_ONLY	5	RW	0x0	When enabled only the DTCM and uncached memory are accessible by load/store operations; accesses to non DTCM cached memory cause a DATA_CACHE_DISABLED exception. If no DTCM is present this bit is reserved and always reads zero.
Reserved	[7:6]	RO	0x0	Reserved
DBREAK_ENABLE	8	RW	0x0	1: data breakpoints are enabled 0: data breakpoints are disabled
IBREAK_ENABLE	9	RW	0x0	1: instruction breakpoints are enabled 0: instruction breakpoints are disabled
Reserved	[11:10]	RO	0x0	Reserved
DEBUG_MODE	12	RW	0x0	1: the core is in debug mode 0: the core is not in debug mode
Reserved	[31:13]	RO	0x0	Reserved

Note: Refer to *Chapter 3: Operating modes on page 23* for the combined behavior of the USER_MODE and DEBUG_MODE bits.

4.4.2 PSW access

The PSW can be read as a control register. See *Section 4.5: Control registers on page 29*.

The core can atomically update bits in the PSW using the **pswmask** operation. This operation returns the unmodified value.

The PSW can also be updated by means of an **rfi** operation. The core uses this operation to restore the context when exiting a trap handler. The state updated by **rfi** is defined in *Section 10.7.3: Restoring execution state on page 90*.

4.5 Control registers

Additional architectural state is held in a number of memory mapped control registers, see *Chapter 15: Control registers on page 145*. These registers include support for traps and for memory protection.

The architectural state saved and restored by a trap handler is defined in *Section 10.7: Saving and restoring execution state on page 87*.

4.6 Atomic address

The core possesses a lock address register. This register is used to lock a resource in a hardware multi-threaded or multi-processor system. The register is described in full in *Chapter 13: Multi-processor and multi-threading support on page 132*.

4.7 Context saving

A full context saving software routine saves the information listed below.

- The contents of control registers. Note, however, the following qualifications.
  - The current PSW is not saved as it relates to the handler; the saved PSW is stored in the SAVED_PSW register.
  - The UTLB contents are not memory mapped, and must be accessed using the TLB_INDEX register, see *Section 11.4.3: TLB_INDEX register on page 98*.
  - If the trap handler is in supervisor mode, control registers only accessible in debug mode cannot be saved. If the trap handler is in debug mode then all control registers can be saved.
  - Control registers that are read only in all operating modes are not saved.
  - The lock bit from the ATOMIC_LOCK register is automatically cleared on a context switch, and the shadow lock bit indicates whether the interrupted context had the lock set, see *Chapter 13: Multi-processor and multi-threading support on page 132*. As the ATOMIC_LOCK and associated ATOMIC_ADDRESS registers are read only they are not saved as part of the context.
- The contents of all branch registers.

  The branch registers must be transferred to general purpose registers using **mov** operations before being stored.
- The contents of all general purpose registers.

5 Bundling rules

The ST240 has a defined set of rules that govern how individual syllables can be bundled together into a single instruction word. Many of these rules are required by the implementation. If the ST240 encounters a bundle which fails to follow the bundling rules, it raises an ILL_INST exception.

Providing the rules listed below are followed, operations can be freely grouped within a bundle. The introduction to *Chapter 2: Execution units on page 18* provides the maximum number of each operation type allowed in a bundle for the current implementation. For example, up to four ALU operations can be included in a single bundle, but there can be no more than one load/store operation or one branch operation (there may be one of each).

5.1 Architectural bundling rules

These restrictions are independent of the ST240 implementation and will therefore remain valid for any future implementation of the ST240 core.

- All syllables must be valid operations or immediate extensions.
- A bundle must have no more than one stop bit set; the stop bits of all but one syllable must be zero.
- Unused opcode fields must be set to zero (this includes the reserved bit 30).
- Immediate extensions must be associated with an operation that is in the same bundle and has an immediate format that can be extended.
- There can be no more than one immediate extension associated with a single operation.
- A privileged operation can only be executed in supervisor or debug mode. (Privileged operations are **rfi** and **pswmask**.)
- The **sbrk** and **dbgsbrk** operations must have the stop bit set.
- Destination registers in a bundle must be unique, that is, can be referenced no more than once in a bundle. (The only exception is R0.)
- The operations **prginsadd, prginsadd.l1, prginsset**, **prginsset.l1, syscall** and **retention** must be the only operation in a bundle.

5.2 Implementation-specific bundling rules

These bundling rules are a consequence of the current ST240 implementation and may change in later implementations.

- A bundle can have no more than four syllables. (A bundle with four zero stop bits is therefore illegal, as it implies a bundle with more than four syllables.)
- A **br**, **brf**, **call**, **rfi**, **return** or **goto** operation must appear as the first syllable of a bundle.
- Multiply operations must appear at odd word addresses.
- **addf.n**, **subf.n**, **convfi.n** and **convif.n** must appear at even word addresses.
- A single bundle may not contain an **addf.n**, **subf.n**, **convfi.n** or **convif.n** operation at a quad word aligned address and an operation requiring the load store unit at an odd word address.
- Immediate extensions must appear at even word addresses.
- There can only be one operation requiring the load/store unit in each bundle. The operations concerned here are **div, divu, rem, remu, flushadd, flushadd.l1, invadd, invadd.l1, prgadd, prgadd.l1, prgset, prginsadd, prginsset, sync**, **syncins, pswmask, rfi, ldb, ldbu, ldbc, ldbuc, ldh, ldhu, ldhc, ldhuc, ldw, ldwc, ldl, ldlc, stb, stbc, sth, sthc, stw, stwc, stl, stlc, pft, pftc, ldwl, stwl, waitl,** and **wmb**.
- Operations with a latency of more than one cycle must not have R63 as a destination register, with the exception of **ldw** and **ldwc**.

5.3 Other restrictions

All branch target addresses and trap vectors must point to the first syllable of a bundle. Any attempt to jump to a syllable that is not the first in a bundle has undefined results.

6 Execution pipeline and latencies

This chapter describes the architecturally visible pipeline and operation latencies of the ST240.

6.1 ST240 pipeline

The ST240 implementation uses a 6-stage pipeline (including three execution stages) with the stages listed in *Table 4*.

Table 4. ST240 pipeline stages

Stage	Abbreviation	Description
Fetch	F	Syllables are fetched from the instruction cache and formed into bundles.
Decode	D	Bundles are decoded and branches are executed.
Read	R	The register files are read.
Execute 1	E1	The first cycle of operation execution.
Execute 2	E2	The second cycle of operation execution.
Writeback	WB	The architectural state is updated.

The following aspects of the pipeline are architecturally visible for performance only:

- operation latencies
- bypassing

The following aspects of the pipeline are architecturally visible and affect the correctness of programs:

- **stw**/**stwc** to **rfi** usage restrictions
- branch stalls
- speculative link register usage restrictions

6.2 Operation latencies and bypassing

The execution pipeline is three cycles long and comprises three stages E1, E2 and WB. All operations begin in E1. Operands are read or bypassed to an operation at the start of E1. All results are written to the register file at the end of WB.

The execution pipeline allows operations to execute for up to three cycles. Some three-cycle operations (such as load, divide and remainder operations) may take longer than three cycles to complete, requiring the pipeline to stall until they have completed. The results of operations that complete earlier than WB are available for bypassing as operands to subsequent operations, though strictly speaking the operation is not complete until the end of the WB stage. The architectural state is updated at WB.

The time taken for an operation to produce a result is called the *operation latency*. Latencies fall into three categories:

- arithmetic operations that take one to three cycles depending upon the operation, see *Chapter 7: Arithmetic operations on page 36*
- logical operations that take one cycle
- load/store operations that take three cycles and may stall the pipeline if more cycles are required

The latency for every operation is specified in *Chapter 24: Instruction set on page 222*.

The pipeline is designed to implement efficiently the serial execution of instruction code, see *Chapter 22: Execution model on page 199*.

6.3 Interlocks and stw/stwc to rfi usage restrictions

The ST240 provides operation latency interlock checking. This enforces the latency between all operations by stalling the pipeline, with the following exceptions:

- **stw**/**stwc** to SAVED_PSW to **rfi**
- **stw**/**stwc** to SAVED_PC to **rfi**
- **stw**/**stwc** to SAVED_SAVED_PSW to **rfi**
- **stw**/**stwc** to SAVED_SAVED_PC to **rfi**

In the cases listed above, the software must ensure that the control register has been updated before executing the **rfi**. It does this by allowing four cycles between the **stw**/**stwc** and the **rfi**. This can be achieved by issuing four bundles (or less if any of those bundles cause interlock or branch stalls). The number of bundles will increase for future implementations that have a longer pipeline length.

For all other cases, the ST240 automatically stalls the pipeline to uphold the internal latency constraints. For optimal machine usage, code should be scheduled to minimize pipeline stalls.

6.4 Branching and branch stalls

The ST240 has no penalty for not-taken branches and stalls for one cycle when a branch is taken. There may be a further stall caused by the destination bundle of a branch crossing an instruction cache line boundary. Branching also has the effect of clearing any unexecuted operations from the instruction fetch logic. See *Section 12.4.2: Instruction fetch on page 116* for more details.

6.5 Link register restrictions

The core uses a speculative link register (SLR) for performance optimization. The SLR holds a copy of possible future updates in R63. The operations that use the SLR are **rfi**, **call $r63, goto $r63** and **return $r63**.

The SLR is not a true copy of R63. To compensate for this, all trap handlers must execute the macro **mov $r63 = $r63** before executing any of the SLR dependant operations listed above.

There is a four cycle latency between a write to R63 and an SLR dependant operation. Interlock stalls are enforced to guarantee this.

A number of operations in the implementation cannot update R63 for efficiency reasons. R63 cannot be updated by:

- any 2 cycle operation
- any 3 cycle operation except **ldw** and **ldwc**

6.6 Operations that empty the pipeline

As state is stored within the pipeline, some changes require that the pipeline is emptied to ensure coherency. For example, the ST240 pipeline needs to be emptied to ensure that UTLB and PSW updates take effect. (For the recommended sequence for UTLB updates, see *Section 11.6.2: TLB coherency on page 105*).

The following operations cause the pipeline to be emptied:

- **rfi**
- **pswmask**
- **syncins**

7 Arithmetic operations

This chapter provides a description of arithmetic operations on the ST240.

7.1 Overview

The ST240 instruction set contains a large number of arithmetic operations including floating point, saturating, fractional, SIMD and joint multiply-add operations. In some cases, these functions overlap; for instance, **mulfracadds.ph**, which is SIMD, saturating and a joint multiply-add operation.

All SIMD operations are described in *Chapter 9: SIMD operations on page 64*.

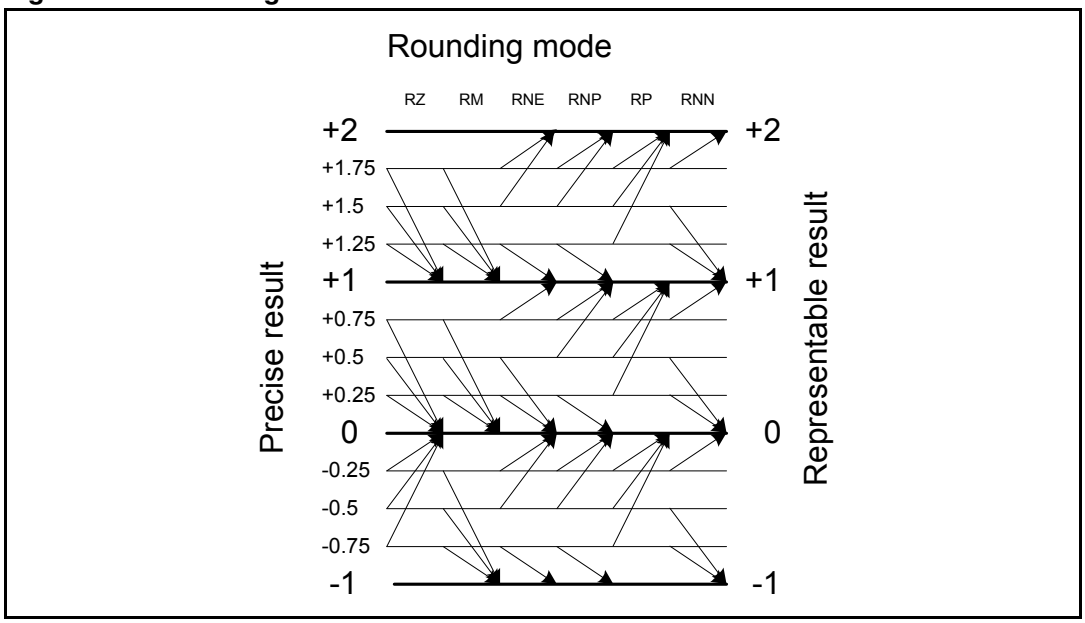
7.1.1 Rounding

A number of the ST240 operations involve rounding. Six rounding modes are provided:

Round towards Zero (RZ)	The result is the representable value closest to and no greater in magnitude than the infinitely precise result. This is equivalent to no rounding for an unsigned number.
Round towards Minus (RM)	The result is the representable value closest to and no greater than the infinitely precise result. This is also known as round towards negative infinity. This is equivalent to no rounding for a signed or unsigned number.
Round to Nearest Even (RNE)	The result is the representable value closest to the infinitely precise result, except when the infinitely precise result lies exactly between two representable values. When this occurs the result is the representable value which has a zero as its least significant bit.
Round to Nearest Positive (RNP)	The result is the representable value closest to the infinitely precise result, except when the infinitely precise result lies exactly between two representable values. When this occurs the result is the greater representable value.
Round towards Positive (RP)	The result is the representable value closest to positive infinity. This is also known as round towards positive infinity.
Round to Nearest Negative (RNN)	The result is the representable value closest to the infinitely precise result, except when the infinitely precise result lies exactly between two representable values. When this occurs the result is the lesser representable value.

Figure 4 illustrates the different behaviors of these rounding modes.

Figure 4. Rounding modes



For many applications RNE is the most useful rounding mode, as there is no directional bias applied to the result. Some SIMD operations are available with different rounding modes (**shr.ph**, **shrrnp.ph**, **shrrne.ph**) as this allows efficient usage for different algorithms.

Table 5. Operations and rounding modes

operation \ rounding mode	RZ	RM	RNE	RNP	RP	RNN
addf.n, **subf.n**, **convif.n**, **convfi.n**, **mulf.n**			X			
mul64h, **mul64hu**		X				
mulfrac				X		
mulfracrne.ph			X			
mulfracrm.ph		X				
shr.ph		X				
shrrnp.ph				X		
shrrne.ph			X			
avgu.pb(1)	X				X	
avg4u.pb(1)	X			X	X	X

1. This operation has a programmable rounding mode

7.1.2 Operand types

The ST240 instruction set supports the following operand formats:

- 32-bit signed and unsigned integers
- 32-bit signed 1.31 fractional integers
- 32-bit single precision floating point
- 16-bit packed signed and unsigned integers
- 16-bit 1.15 fractional signed integers
- 8-bit packed unsigned integers

Except where noted otherwise.

- All 32-bit integer operations may be applied to 32-bit signed or unsigned integers, or 32-bit signed 1.31 fractional integers.
- All 16-bit integer SIMD operations may be applied to16-bit signed or unsigned integers, or 16-bit signed 1.15 fractional integers.

SIMD operands

Table 6 shows how the SIMD (packed) data types map to a 32-bit operand.

Table 6. SIMD data types

Four packed 8-bit values

8-bit data bit field 3		8-bit data bit field 2		8-bit data bit field 1		8-bit data bit field 0	
31	24	23	16	15	8	7	0

Two packed 16-bit values

16-bit data bit field 1		16-bit data bit field 0	
31	16	15	0

Fractional operands

The ST240 supports both 16-bit and 32-bit fractional operands. This is a two's complement representation in which the sign bit has weight -1 and the other bits therefore have positive fractional weights; hence the range [-1,+1) shown in *Table 7*.

Table 7. Fractional representation

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

Fractional data formats are described with two numbers separated by a '.' (full stop). These specify the number of bits before (including the sign bit) and after the binary point. The 16-bit fractional numbers which appear in the specification use the 1.15 format and 32-bit fractional numbers use the 1.31 format.

Floating point operands

The floating point operands are in IEEE754 single precision format. Only scalar floating point is available, no packed types are supported.

7.1.3 SIMD operation naming

A consistent naming convention is used for the SIMD operations so that, with knowledge of some simple rules, the semantics of most operations can be deduced.

SIMD operation names are all of the form:

*<opn>***s**?(**u**|**us**|)?**.p**(**h**|**b**)(|**h**|**l**)

That is, the following suffixes are added, in the specified order, to the base name of the operation, *<opn>*, being performed.

- An optional **s** indicates that saturation is performed.
- A **u** or **us** before the full stop indicates the operation is unsigned, **u** indicates the operation is unsigned, **us** indicates that the first source operand is unsigned, but the second and result operands are signed. Only one operation requires this latter form: **muladdus.pb**.
- The suffix, **.p**, indicates that this is a SIMD (packed) operation.
- The **.p** suffix is always followed either by an **h** or **b** to indicate the size of the packed objects in the operation, 16-bit or 8-bit respectively. For operations where the result elements differ in size from the source elements the width is chosen to be consistent with the operation. For example **packs.ph**, packs two 32-bit values into a register containing two saturated 16-bit results, while **unpacku.pbl** unpacks unsigned 8-bit values into 16-bit ones.
- An optional **h** or **l** is appended to indicate that the complete result is 64-bits wide and must be returned using two separate operations, **l** for the lower 32-bits, **h** for the upper 32-bits.

Table 8 provides examples.

Table 8. Examples of SIMD operation naming

Operation	Description
add.ph	Packed 16-bit addition. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
subs.ph	Packed 16-bit signed subtraction with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
cmpgtu.pb	Packed 8-bit unsigned greater than.
muladdus.pb	Unsigned 8-bit integer multiplied by signed 8-bit integer value and add across.
shuff.pbh	Packed 8-bit shuffle returning high result.

Note: Scalar operations do not follow this naming scheme.

7.2 Multiplication operations

The ST240 instruction set includes a wide range of multiplication operations. These are listed in *Table 9*.

Table 9. Multiplication summary table

Operation	Syntax	Latency	Description
mul32	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	3	32x32-bit signed multiplication, returns lower 32 bits of the intermediate 64-bit result. Operands are signed or unsigned integers.
mul64h	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	3	32x32-bit signed multiplication, returns upper 32 bits of the intermediate 64-bit result. Operands are signed integers.
mul64hu	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	3	32x32-bit signed multiplication, returns upper 32 bits of the intermediate 64-bit integers.
mulhh	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Upper-half-word by upper-half-word signed multiplication. Operands are 16-bit signed integers.
mulhhu	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Upper-half-word by upper-half-word unsigned multiplication. Operands are 16-bit unsigned integers.
mullh	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Half-word by upper-half-word signed multiplication. Operands are 16-bit and 32-bit signed integers.
mullhu	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Half-word by upper-half-word unsigned multiplication. Operands are 16-bit and 32-bit unsigned integers.
mulll	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	3	Half-word by half-word signed multiplication. Operands are 16-bit signed integers.
mullllu	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	3	Half-word by half-word unsigned multiplication. Operands are 16-bit unsigned integers.
mul.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit multiplication. Operands may be signed or unsigned 16-bit integers.
muladd.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit signed multiplication and add across with saturation. Operands are signed 16-bit integers.
mulh	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Word by upper-half-word signed multiplication. Operands are 16-bit and 32-bit signed integers.
mull	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Word by half-word signed multiplication. Operands are 16-bit and 32-bit signed integers.

Table 9. Multiplication summary table (continued)

Operation	Syntax	Latency	Description
mulfrac	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	3	32-bit fractional multiplication with round nearest positive and saturation. Operands are fractional 1.31 format.
mulfracrne.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit fractional multiplication with round nearest even and saturation. Operands are fractional 1.15 format.
mulfracrm.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit fractional multiplication with round minus and saturation. Operands are fractional 1.15 format.
mulfracadds.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit signed fractional multiplication and add across with saturation. Operands are fractional 1.15 fractional format.
muladdus.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Unsigned 8-bit integer multiplied by signed 8-bit integer value and add across.
mulf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	IEEE754 format single precision floating point multiplication.

7.3 Addition and subtraction operations

The ST240 instruction set includes a wide range of addition and subtraction operations. These are listed in *Table 10*.

Table 10. Addition and subtraction summary table

Operation	Syntax	Latency	Description
Scalar operations			
add	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Scalar 32-bit addition. Operands may be signed or unsigned integers or fractional 1.31 format.
addpc	$R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Scalar 32-bit addition of immediate value and virtual PC of the current bundle.
sub	$R_{DEST} = R_{SRC2}, R_{SRC1}$ $R_{DEST} = I_{SRC2}, R_{SRC1}$	1	Scalar 32-bit subtraction. Operands may be signed or unsigned integers or fractional 1.31 format.
sh1add	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Shift the first operand left one place and perform a 32bit scalar addition with the second operand. Operands may be signed or unsigned integers or fractional 1.31 format.
sh2add	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Shift the first operand left two places and perform a 32bit scalar addition with the second operand. Operands may be signed or unsigned integers or fractional 1.31 format.

Table 10. Addition and subtraction summary table (continued)

Operation	Syntax	Latency	Description
sh3add	$R_{DEST} = R_{SRC1}, R_{SRC}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Shift the first operand left three places and perform a 32bit scalar addition with the second operand. Operands may be signed or unsigned integers or fractional 1.31 format.
addcg	$R_{DEST},B_{DEST} = R_{SRC1}, R_{SRC2},B_{SCOND}$	1	32-bit scalar addition with carry input and generate a carry output. Operands may be signed or unsigned integers or fractional 1.31 format.
Saturating scalar operations			
adds	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Scalar 32-bit addition with saturation. Operands may be signed or unsigned integers or fractional 1.31 values.
addso	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Indicates whether an **adds** operation with the given input operands causes a saturation.
subs	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Scalar 32bit subtraction with saturation. Operands may be signed integers or fractional 1.31 format.
subso	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Indicates whether a **subs** operation with the given input operands causes a saturation
sh1adds	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Shift the first operand left one place and perform a 32bit scalar addition with the second operand. Operands may be signed or unsigned integers or fractional 1.31 format.
sh1addso	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Indicates whether a **sh1adds** operation with the given input operands causes a saturation
sh1subs	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Shift Rsrc1 left one place, saturate to 32 bits and then subtract from Rsrc2 and saturate again. Operands may be signed integers or fractional 1.31 format.
sh1subso	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Indicates whether a **sh1subs** operation with the given input operands causes a saturation
SIMD operations			
add.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit addition. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
adds.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed addition with saturation. Operands may be 16-bit scalar or fractional 1.15 format.

Table 10. Addition and subtraction summary table (continued)

Operation	Syntax	Latency	Description
sub.ph	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Packed 16-bit subtraction. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
subs.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed subtraction with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
Floating point operations			
addf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	IEEE754 format single precision floating point add
subf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	IEEE754 format single precision floating point subtract

7.4 Shift operations

The ST240 instruction set includes a wide range of arithmetic shift operations. These are listed in *Table 11*. The logical shifts are covered in *Chapter 8: Logical operations on page 59*.

Table 11. Shift summary table

Operation	Syntax	Latency	Description
Scalar operations			
shl	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Scalar 32-bit arithmetic or logical left shift. Operands may be signed or unsigned integers or fractional 1.31 format.
shr	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Scalar 32-bit arithmetic right shift. Operands may be signed integers or fractional 1.31 format.
shru	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Scalar 32-bit logical right shift. Operands are unsigned integers.
shrrnp	$R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Scalar 32-bit arithmetic right shift with round nearest positive rounding. Operands may be signed or unsigned integers or fractional 1.31 format.
Saturating scalar operations			
shls	$R_{DEST} = R_{SRC1}, R_{SRC2}$	2	Scalar 32-bit left shift with saturation. Operands may be signed integers or fractional 1.31 format.
shlso	$R_{DEST} = R_{SRC1}, R_{SRC2}$	2	Indicates whether a **shls** operation with the same input operands would have saturated

Table 11. Shift summary table (continued)

Operation	Syntax	Latency	Description
SIMD operations			
shl.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Packed 16-bit left shift. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
shr.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Packed 16-bit arithmetic right shift. Operands may be signed 16-bit integer or fractional 1.15 format.
shrrne.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift right with round nearest even rounding. Operands may be signed 16-bit integer or fractional 1.15 format.
shrrnp.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift right with round nearest positive rounding. Operands may be signed 16-bit integer or fractional 1.15 format.
shls.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift left with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

The rounding modes used for the shifts with rounding are described in *Table 7.1.1: Rounding on page 36*.

7.5 Comparison operations

The ST240 instruction set includes a full set of scalar comparisons. They are not all listed in this section as their behavior can be determined from their names. Refer to *Chapter 24: Instruction set on page 222* for a complete list.

Table 12 lists the SIMD and floating point comparisons. Note that two macros are included for floating point comparisons. Please refer to *Section 24.4: Macros on page 226*.

Table 12. SIMD and floating point comparison summary table

Operation	Syntax	Latency	Description
SIMD operations			
cmpeq.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 8-bit test for equality.
cmpgtu.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 8-bit unsigned greater than.
cmpeq.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit test for equality.
cmpgt.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed greater than.
max.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed maximum. Operands may be signed 16-bit integers or fractional 1.15 format.

Table 12. SIMD and floating point comparison summary table (continued)

Operation	Syntax	Latency	Description
min.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed minimum. Operands may be signed 16-bit integers or fractional 1.15 format.
Floating point operations			
cmpeqf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	IEEE754 format single precision floating point equality comparison.
cmpgtf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	IEEE754 format single precision floating point greater than comparison.
Floating point macros			
cmpltf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	IEEE754 format single precision less than comparison.
cmplef.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	IEEE754 format single precision less than or equal to comparison.

Floating point comparisons are described further in *Section 7.8.3: Floating point comparison operations on page 53*.

7.6 Saturating arithmetic operation usage

The ST240 provides a number of saturating arithmetic operations. These are to enhance the performance of telecommunications algorithms such as EFR and AMR. The operations are efficiently used to implement functions defined by ETSI. Refer to ETSI documentation for a description of these functions.

Saturating arithmetic operations may be required to indicate arithmetic overflow, as required by the application. Therefore each scalar saturating arithmetic operation has two versions:

- **Saturating**: return numerical result from saturating operation into a register
- **Overflowing**: return overflow flag from saturating operation into a register; return a boolean value represented by the integer value 0 or 1

7.6.1 Saturating operation behavior

The **saturating** operations perform a monadic or dyadic operation on the input operands, and limit the result to a fixed output range. The **overflowing** operations report whether an overflow condition occurred. To obtain both results, issue both forms of the operation.

The following functions are added to the operation notation specification, where $n > 0$:

- $Saturate_n()$
- $Overflow_n()$

See *Chapter 23: Specification notation on page 202* for the definitions of these functions.

See *Table 14* for the value of *n* for each operation.

All operations have the following properties:

- no laning restrictions
- take one or two register operands and write to a register
- cannot take immediate operands

A summary of saturating operations is provided in *Table 13*.

Table 13. Saturating operations summary

Operation	Syntax	Latency	Description
Scalar saturating operations			
adds	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Scalar 32-bit addition with saturation. Operands may be signed or unsigned integers or fractional 1.31 values.
subs	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Scalar 32-bit subtraction with saturation. Operands may be signed integers or fractional 1.31 value.
sh1adds	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Shift the first operand left one place and perform a 32-bit scalar addition with the second operand. Operands may be signed or unsigned integers or fractional 1.31 format.
sh1subs	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Shift Rsrc1 left one place, saturate to 32 bits and then subtract from Rsrc2 and saturate again. Operands may be signed integers or fractional 1.31 format.
sats	$R_{DEST} = R_{SRC1}$	1	Saturate from 32-bit scalar to 16-bit scalar. The operand is a signed integer or is fractional 1.31 format.
shls	$R_{DEST} = R_{SRC1}, R_{SRC2}$	2	Scalar 32-bit left shift with saturation. Operands may be signed integers or fractional 1.31 format.
Scalar overflowing operations			
addso	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Indicates whether an **adds** operation with the given input operands causes a saturation.
subso	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Indicates whether a **subs** operation with the given input operands causes a saturation.
sh1addso	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Indicates whether a **sh1adds** operation with the given input operands causes a saturation.
sh1subso	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Indicates whether a **sh1subs** operation with the given input operands causes a saturation.
satso	$R_{DEST} = R_{SRC1}$	1	Indicate if a **sats** operation with the same operand would have saturated.

Table 13. Saturating operations summary (continued)

Operation	Syntax	Latency	Description
shlso	$R_{DEST} = R_{SRC1}, R_{SRC2}$	2	Indicates whether a **shls** operation with the same input operands would have saturated.
SIMD saturating operations			
mulfracrm.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit fractional multiplication with round minus and saturation. Operands are fractional 1.15 format.
mulfracrne.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit fractional multiplication with round nearest even and saturation. Operands are fractional 1.15 format.
mulfracadds.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit signed fractional multiplication and add across with saturation. Operands are fractional 1.15 fractional format.
adds.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed addition with saturation. Operands may be 16-bit scalar or fractional 1.15 format.
subs.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed subtraction with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

7.6.2 Saturating operations usage for implementation of ETSI functions

The saturating operations are listed in *Table 14*, together with the ETSI functions that they are used to implement.

Table 14. Saturating operations usage for ETSI functions

Operation	Saturation width	ETSI function
Scalar saturating operations		
adds	32	**L_add**()
subs	32	**L_sub**()
sh1adds	32	Component of **L_mac**()
sh1subs	32	Component of **L_msu**()
sats	16	**saturate**()
shls	32	Component of **L_shl**(), immediate form included as most shifts in AMR-NB are constant.
Scalar overflowing operations		
addso	32	**L_add**()
subso	32	**L_sub**()
sh1addso	32	Component of **L_mac**()

Table 14. Saturating operations usage for ETSI functions (continued)

Operation	Saturation width	ETSI function
sh1subso	32	Component of **L_msu**()
satso	16	**saturate**()
shlso	32	Component of **L_shl**(), immediate form included as most shifts in AMR-NB are constant
SIMD saturating operations		
mulfracrm.ph	16	SIMD form of **mult()**
mulfracrne.ph	16	This does **not** map to an ETSI function.
mulfracadds.ph	16	Performs (**L_add**(**L_mult**(operand1$_{low}$,operand2$_{low}$),**L_mult**(operand 1$_{high}$,operand2$_{high}$)) - see *Section 7.6.3*.
adds.ph	16	SIMD form of **add**()
subs.ph	16	SIMD form of **sub**()

7.6.3 mulfracadds.ph usage

The **mulfracadds.ph** operation does not map directly to an ETSI function, but is used to improve the performance of loops containing calls to **L_mac**() and **L_msu**(). The transformation required to recode a loop to use **mulfracadds.ph** may cause saturation behavior to vary, as the calculations need to be refactored. Therefore any use of **mulfracadds.ph** may require conformance checks to be made. An example of a loop using **L_mac**() is:

```
int s; # 32-bit type
short a[1024], b[1024]; # 16-bit types
....
s=0;

for (i=0;i<1024;i++) {
   s = L_mac(s,a[i],b[i]);
}
```

The loop is modified to read packed operands and refactored to allow use of **mulfracadds.ph** as follows:

```
s=0;
int a_val, b_val, s1, s2, s3;

for (i=0;i<1024;i+=2) {
   a_val = *(int *)(a+i); # read pair of operands
   b_val = *(int *)(b+i); # be careful with alignment
   s1 = L_mult((short)a_val,(short)b);
   s2 = L_mult((short)(a_val>>16),(short)(b>>16));
   s3 = L_add(s1, s2);
   s = L_add(s3, s);
}
```

mulfracadds.ph is now used:

```
s=0;
int a_val, b_val, s3;

for (i=0;i<1024;i+=2) {
   a_val = *(int *)(a+i); # read pair of operands
   b_val = *(int *)(b+i); # be careful with alignment
   s3 = __st200mulfracadds_ph(a_val, b_val);
   s = L_add(s, s3);
}
```

Loops based upon **L_msu**() use **L_sub**(s, s3) function to assign to s instead of **L_add**(s, s3).

7.7 SIMD arithmetic operations usage

All the arithmetic SIMD operations are summarized in *Table 15*. For a listing of logical SIMD operations please refer to *Chapter 8: Logical operations on page 59*.

Table 15. SIMD arithmetic operations summary

Operation	Syntax	Latency	Description
16-bit arithmetic operations			
abss.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed absolute with saturation. Operands may be signed 16-bit integers or fractional 1.15 format.
add.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit addition. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format
adds.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed addition with saturation. Operands may be 16-bit scalar or fractional 1.15 format.
cmpeq.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit test for equality
cmpgt.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed greater than
max.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed maximum. Operands may be signed 16-bit integers or fractional 1.15 format.
min.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed minimum. Operands may be signed 16-bit integers or fractional 1.15 format.

Table 15. SIMD arithmetic operations summary (continued)

Operation	Syntax	Latency	Description
mul.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit multiplication. Operands may be signed or unsigned 16-bit integers.
muladd.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit signed multiplication and add across with saturation. Operands are signed 16-bit integers.
mulfracadds.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit signed fractional multiplication and add across with saturation. Operands are fractional 1.15 fractional format.
mulfracrm.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit fractional multiplication with round minus and saturation. Operands are fractional 1.15 format.
mulfracrne.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit fractional multiplication with round nearest even and saturation. Operands are fractional 1.15 format.
shl.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Packed 16-bit left shift. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
shls.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift left with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
shr.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Packed 16-bit arithmetic right shift. Operands may be signed 16-bit integer or fractional 1.15 format.
shrrne.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift right with round nearest even rounding. Operands may be signed 16-bit integer or fractional 1.15 format.
shrrnp.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift right with round nearest positive rounding. Operands may be signed 16-bit integer or fractional 1.15 format.
sub.ph	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Packed 16-bit subtraction. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

Table 15. SIMD arithmetic operations summary (continued)

Operation	Syntax	Latency	Description
subs.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed subtraction with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
8-bit arithmetic operations			
absubu.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 8-bit unsigned absolute difference
avgu.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	1	Packed 8-bit unsigned average with selectable rounding mode (round zero or round positive).
avg4u.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	2	Packed 8-bit unsigned 4-way average with selectable rounding mode (round zero, round nearest negative, round nearest positive or round positive).
cmpeq.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 8-bit test for equality.
cmpgtu.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 8-bit unsigned greater than.
muladdus.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Unsigned 8-bit integer multiplied by signed 8-bit integer value and add across.
sadu.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	2	Sum of absolute differences on packed unsigned 8-bit values.

7.8 Floating point operations

Low latency single precision floating point is available, using the IEEE754 number format.

7.8.1 Summary of floating point operations and macros

Table 16 summarizes the floating point operations and macros.

Table 16. Summary of floating point operations

Operation	Syntax	Latency	Description
Floating point operations			
convif.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Signed integer to IEEE754 format single precision floating point conversion.
convfi.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	IEEE754 format single precision floating point to signed integer conversion.
addf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	IEEE754 format single precision floating point add.

Table 16. Summary of floating point operations (continued)

Operation	Syntax	Latency	Description
subf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	IEEE754 format single precision floating point subtract.
mulf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	IEEE754 format single precision floating point multiplication.
cmpeqf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	IEE754 format single precision floating point equality comparison.
cmpgtf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	IEE754 format single precision floating point greater than comparison.
Floating point macros			
cmpltf.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	IEE754 format single precision less than comparison.
cmplef.n	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	IEE754 format single precision less than or equal to comparison.

7.8.2 IEEE754 specification limitations

The floating point operations do not follow the full IEEE754 specification. The differences from the standard behavior are listed in the following section.

Rounding Modes

Only the round to nearest even rounding mode is available for **addf.n, subf.n, mulf.n convif.n**.

Only the round to zero mode is available for **convfi.n**.

See *Section 7.1.1: Rounding on page 36* for more information about rounding modes.

Denormalised number handling

Denormalised operands are operated upon as if they are signed zeroes (the sign of the denormalised operand is retained). If the result of an operation is required by the IEEE754 standard to produce a denormalised result of either sign then the result produced is +0 instead.

Exceptions

These operations do not produce status flags and do not raise any exceptions.

Positive and negative zero

These operations never produce -0; +0 is produced instead. For example: multiplying the negative minimum normalized number by the positive minimum normalized number returns +0 whereas the IEEE754 standard requires -0.

IEEE754 specification interpretation

In some cases the IEEE754 specification is not fully specific. The choices made by this architecture to cover these cases is listed in the following sections.

Not a number handling

This architecture does not implement propagation of NaNs. Whenever a NaN is produced, it will be as shown in *Table 17*.

Table 17. NaN value

Name	Value	Description
GENERATED_NAN	0x7FFFFFFF	Single precision qNaN

sNaNs (signalling NaNs) and qNaNs (quiet NaNs) are distinguished by looking at the top bit of the mantissa field (bit 22). This bit is set for a qNaN and clear for an sNaN.

Conversions of unrepresentable floating numbers to integers

Numbers such as NaNs, infinities and out of range numbers cannot be correctly converted to integers. The numerical values produced for signed integers are as shown in *Table 18*.

Table 18. Unrepresentable conversion value for signed integers

Name	Value	Description
UNREP_SIGNED_ POS	0x7FFFFFFF	Out of range positive numbers including positive NaNs
UNREP_SIGNED_ NEG	0x80000000	Out of range negative numbers including negative NaNs

Table 19 shows the results for unsigned integers for future architectures that support these.

Table 19. Unrepresentable conversion value for unsigned integers

Name	Value	Description
UNREP_UNSIGNED_ POS	0xFFFFFFFF	Out of range positive numbers including positive NaNs
UNREP_UNSIGNED_ NEG	0x00000000	All negative numbers

Integers

No rounding is performed when integers are produced. Therefore the round towards zero mode is used.

Tininess detection

This architecture detects tininess before rounding. This affects future IEEE754 operations that use the UNDERFLOW flag and does not effect any existing operations.

Infinities, NaNs and zeroes

Infinities, NaNs and zeroes are handled as defined by the IEEE754 specification except as noted in *Positive and negative zero on page 52*.

7.8.3 Floating point comparison operations

The behavior is as defined by IEEE754, except that no status flags are produced, so no exceptions can be thrown.

7.9 Fractional arithmetic operations

Many scalar operations may be reused to operate on fractional operations. Specific multiplication operations must be used for fractional arithmetic. For addition and subtraction either the non saturating or the saturating forms may be used.

Table 20 lists all fractional arithmetic operations.

Table 20. Fractional operations summary table

Operation	Syntax	Latency	Description
Fractional multiplication operations			
mulfrac	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	3	32-bit fractional multiplication with round nearest positive and saturation. Operands are fractional 1.31 format.
mulfracrne.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit fractional multiplication with round nearest even and saturation. Operands are fractional 1.15 format.
mulfracrm.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit fractional multiplication with round minus and saturation. Operands are fractional 1.15 format.
mulfracadds.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit signed fractional multiplication and add across with saturation. Operands are fractional 1.15 fractional format.
Fractional addition and subtraction operations			
add	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	32-bit integer addition. Operands may be signed or unsigned 32-bit scalar or signed fractional 1.31 format.
sub	$R_{DEST} = R_{SRC2}, R_{SRC1}$ $R_{DEST} = I_{SRC2}, R_{SRC1}$	1	32-bit integer subtraction. Operands may be 32-bit scalar or fractional 1.31 format.
sh1add	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Shift the first operand left one place and perform a 32-bit integer addition with the second operand. Operand types may be 32bit scalar or fractional 1.31 format.
sh2add	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Shift the first operand left two places and perform a 32-bit integer addition with the second operand. Operands may be 32-bit scalar or fractional 1.31 format.
sh3add	$R_{DEST} = R_{SRC1}, R_{SRC}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Shift the first operand left three places and perform a 32-bit integer addition with the second operand. Operands may be 32-bit scalar or fractional 1.31 format.
addcg	$R_{DEST},B_{DEST} = R_{SRC1}$, R_{SRC2},B_{SCOND}	1	32-bit integer addition with carry input and generate a carry output. Operands may be 32-bit scalar or fractional 1.31 format.

Table 20. Fractional operations summary table (continued)

Operation	Syntax	Latency	Description
Saturating fractional operations			
adds	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Integer 32-bit addition with saturation. Operands may be 32-bit scalar or fractional 1.31 format.
addso	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Indicates whether an **adds** operation with the given input operands causes a saturation.
subs	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Integer 32-bit subtraction with saturation. Operands may be 32-bit scalar or fractional 1.31 format.
subso	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Indicates whether a **subs** operation with the given input operands causes a saturation. Operands may be 32-bit scalar or fractional 1.31 format.
sh1adds	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Shift Rsrc1 left one place, saturate to 32 bits and then perform a 32-bit integer addition to Rsrc2 and saturate again. Operands may be 32-bit scalar or fractional 1.31 format.
sh1addso	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Indicates whether a **sh1adds** operation with the given input operands causes a saturation.
sh1subs	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Shift Rsrc1 left one place, saturate to 32 bits and then subtract from Rsrc2 and saturate again. Operands may be 32-bit scalar or fractional 1.31 format.
sh1subso	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Indicates whether a **sh1subs** operation with the given input operands causes a saturation. Operands may be 32-bit scalar or fractional 1.31 format.
sats	$R_{DEST} = R_{SRC1}$	1	Saturate from 32-bit scalar to 16-bit scalar. The operand is a signed integer or is fractional 1.31 format.
satso	$R_{DEST} = R_{SRC1}$	1	Indicate if a **sats** operation with the same operand would have saturated.
shls	$R_{DEST} = R_{SRC1}, R_{SRC2}$	2	Scalar 32-bit left shift with saturation. Operands may be signed integers or fractional 1.31 format.
Fractional scalar shifts			
shl	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Scalar 32-bit arithmetic or logical left shift. Operands may be signed or unsigned integers or fractional 1.31 format.
shr	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Scalar 32-bit arithmetic right shift. Operands may be signed integers or fractional 1.31 format.

Table 20. Fractional operations summary table (continued)

Operation	Syntax	Latency	Description
shrrnp	$R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Scalar 32-bit arithmetic right shift with round nearest positive rounding. Operands may be signed or unsigned integers or fractional 1.31 format.
Fractional SIMD operations			
add.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit addition. Operands may be signed or unsigned 16-bit scalar or fractional 1.15 format.
adds.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed addition with saturation. Operands may be 16-bit scalar or fractional 1.15 format.
sub.ph	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Packed 16-bit subtraction. Operands may be 16-bit scalar or fractional 1.15 format.
subs.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed subtraction with saturation. Operands may be 16-bit signed integer or fractional 1.15 format.
abss.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed absolute with saturation. Operands may be signed 16-bit integers or fractional 1.15 format.
max.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed maximum. Operands may be signed 16-bit integers or fractional 1.15 format.
min.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed minimum. Operands may be signed 16-bit integers or fractional 1.15 format.
shl.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Packed 16-bit left shift. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
shr.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Packed 16-bit arithmetic right shift. Operands may be signed 16-bit integer or fractional 1.15 format.
shrrne.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift right with round nearest even rounding. Operands may be signed 16-bit integer or fractional 1.15 format.
shrrnp.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift right with round nearest positive rounding. Operands may be signed 16-bit integer or fractional 1.15 format.
shls.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift left with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

7.10 Divide and remainder operations

Table 21 lists the integer divide and remainder operations.

Table 21. Divide and remainder operations

Operation	Syntax	Latency[a]	Description
div	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	3	Signed integer division. May stall the pipeline, see definition of IDivIeee(). Operands are signed integers.
divu	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	3	Unsigned integer division. May stall the pipeline, see definition of UIDivIeee(). Operands are unsigned integers.
rem	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	3	Signed integer remainder. May stall the pipeline, see definition of IRemIeee(). Operands are signed integers.
remu	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	3	Unsigned integer remainder. May stall the pipeline, see definition of UIRemIeee(). Operands are unsigned integers.

a. The latency with respect to instruction scheduling is 3 cycles, however additional stall cycles may occur as detailed in the following section.

The divide and remainder operations are restricted to one per bundle as they use load/store resources.

7.10.1 Special cases

Table 22 lists the special cases for divide and remainder operations.

Table 22. Divide and remainder special cases

operation	dividend	divisor	result
div	positive	0	0x7fffffff
div	negative	0	0x80000000
div	0x80000000	0xffffffff	0x80000000[a]
rem	any	0	0
rem	0x80000000	0xffffffff	0
divu	any	0	0xffffffff
remu	any	0	0

a. This result appears to be negative as the result overflows.

No exception is generated for any divide by zero case. If an exception is required then this must be coded in software.

7.10.2 Performance information

The number of cycles taken to execute div/rem operations in this implementation of the ST240 are provided in *Table 23*. Future implementations may have different performance. The variables are defined as follows:

- d is the number of leading zeros in the dividend (if unsigned) or leading sign bits (if signed)
- r is the number of leading zeros in the divisor (if unsigned) or leading sign bits (if signed)

Table 23. cycles taken to execute div/rem operations

Operation	Cycles
div/divu, **rem/remu**	max(21 - (31 + d - r)/2, 5) [(a)]

a. The / in the formula is an integer division.

In all cases where the second operand is non-zero these operations stall the ST240 while iterating. The number of stall cycles is three less than the number of cycles given in *Table 23* and can be measured using the PM_EVENT_DSTALLCYCLES performance monitor. If the second operand is zero then these operations have a three cycle latency.

8 Logical operations

The ST240 instruction set contains a large number of logical operations. They take either scalar or packed operands as inputs. All operations in this category have a single cycle latency.

A sub-set of operations, such as **mov** and **andl**, have forms that write values to the branch registers.

8.1 Scalar logical operations

Table 24 lists all scalar logical operations.

Table 24. Scalar operations summary table

Operation	Syntax	Description
Simple logical primitives		
and	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	Bitwise AND.
andl	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = B_{SRC1}, B_{SRC2}$	Logical AND.
andc	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	Negate operand 1 and then bitwise AND.
nandl	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, I_{SRC2}$ $B_{DEST2} = B_{SRC1}, B_{SRC2}$	Logical NAND.
or	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Bitwise OR.
orl	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, I_{SRC2}$ $B_{DEST2} = B_{SRC1}, B_{SRC2}$	Logical OR.
orc	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	Negate operand 1 and then bitwise OR.
norl	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = B_{SRC1}, B_{SRC2}$	Logical NOR.
xor	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	Bitwise XOR.
slct	$R_{IDEST} = B_{SCOND}, R_{SRC1}, I_{SRC2}$ $R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	Select between registers and immediate operands using boolean value of branch register.
slctf	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	Select between general purpose registers using negated boolean value of branch register.

Table 24. Scalar operations summary table (Continued)

Operation	Syntax	Description
Move operations		
mov	$R_{DEST} = B_{SRC1}$	Move a branch register value into a general purpose register.
	$B_{DEST} = R_{SRC1}$	Move a general purpose register value into a branch register.
	$B_{DEST} = B_{SRC1}$	Move a branch register value into another branch register.
Other logical operations		
extract	$R_{DEST} = R_{SRC1}, I_{SRC2}$	Generalized signed bit field extract for small fields (1-16 bits).
extractu	$R_{DEST} = R_{SRC1}, I_{SRC2}$	Generalized unsigned bit field extract for small fields (1-16 bits).
extractl	$R_{DEST} = R_{SRC1}, I_{SRC2}$	Generalized signed bit field extract for large fields (>=17 bits).
extractlu	$R_{DEST} = R_{SRC1}, I_{SRC2}$	Generalized unsigned bit field extract for large fields (>=17 bits).
rotl	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{IDEST} = R_{SRC1}, I_{SRC2}$	Rotate left.
shl	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{IDEST} = R_{SRC1}, I_{SRC2}$	Scalar 32-bit arithmetic or logical left shift.
shru	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{IDEST} = R_{SRC1}, I_{SRC2}$	Scalar 32bit logical right shift. Operands are unsigned integers. Operands may be signed or unsigned integers or fractional 1.31 format.
sxt	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{IDEST} = R_{SRC1}, I_{SRC2}$	Arbitrary sign extend.
zxt	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{IDEST} = R_{SRC1}, I_{SRC2}$	Arbitrary zero extend.
Operations available as macros		
slctf	$R_{IDEST} = B_{SCOND}, R_{SRC1}, I_{SRC2}$	Byte select between register and immediate operands using negated boolean value in branch register.
convbi	$R_{DEST} = B_{SRC1}$	Perform a boolean conversion on the branch register value and write the result to a general purpose register.
convib	$B_{DEST} = R_{SRC1}$	Perform a boolean conversion on the general purpose register value and write the result to a branch register.
mov	$R_{DEST} = R_{SRC1}$	Move general purpose register value into another general purpose register.
sxtb	$R_{DEST} = R_{SRC1}$	Sign extend byte value.
sxth	$R_{DEST} = R_{SRC1}$	Sign extend half-word value.

Table 24. Scalar operations summary table (Continued)

Operation	Syntax	Description
zxtb	$R_{DEST} = R_{SRC1}$	Zero extend byte value.
zxth	$R_{DEST} = R_{SRC1}$	Zero extend half-word value

8.1.1 Bit extraction operations

The ST240 has four extraction operations (**extract**, **extractu**, **extractl**, **extractlu**). These operations extract a variable number of bits from a selectable bit position within a register. Associated operations are the arbitrary sign and zero extend operations **sxt**, **zxt**.

The extraction operations take two operands:

- an immediate operand to define the bit position and offset to extract
- a register operand on which the extraction is performed

Only bits [8:0] of the immediate operand are used by the extraction operations, so immediate extensions are never required, but may be accidentally inferred if care is not taken.

Danger: If bit 8 is set in the immediate operand, the programmer must sign extend the operand to avoid the ST200 assembler adding an immediate extension of all zeroes.

The extraction operations are listed in *Table 25*.

Table 25. Extract operations

Extract operation	Imm[8:5]	Imm[4:0]	Action
extract	[*length* - 1]	position	extract *length* bits starting from bit *position.* Sign extend result to 32 bits.
extractu	[*length* - 1]	position	extract *length* bits starting from bit *position.* Zero extend result to 32 bits.
extractl	[*length* - 17]	position	extract *length* bits starting from bit *position.* Sign extend result to 32 bits.
extractlu	[*length* - 17]	position	extract *length* bits starting from bit *position.* Zero extend result to 32 bits.

Therefore **extract**(**u**) is used to extract 1 to 16 bits and **extractl**(**u**) is used to extract 17 to 32 bits.

Note: In all cases if (length + position) > 32 then an ILL_INST exception is raised.

8.2 SIMD logical operations

Table 26 lists all SIMD logical operations. SIMD operations are described in *Chapter 9: SIMD operations on page 64*.

Table 26. Packed operations summary table

Operation	Syntax	Description
Simple logical primitives		
slct.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$ $R_{IDEST} = B_{SCOND}, R_{SRC1}, I_{SRC2}$	Byte select between input operands using 4-bit condition code in branch register.
slctf.pb	$R_{IDEST} = B_{SCOND}, R_{SRC1}, I_{SRC2}$	Byte select between input operands using 4-bit condition code in branch register.
Extraction operations		
ext1.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Extract word starting from byte 1.
ext2.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Extract word starting from byte 2.
ext3.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Extract word starting from byte 3.
extl.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	Dynamic extract left operation.
extr.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	Dynamic extract right operation.
Packing operations		
pack.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Pack 4 16-bit values to 8-bit results ignoring upper bits.
packrnp.phh	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Pack high part of 32-bit signed value into 16-bit signed results with round nearest positive.
packs.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Pack 32-bit signed values into 16-bit signed results with saturation.
packsu.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Pack 16-bit signed values into 8-bit unsigned results with saturation.
Shuffle and permute operations		
perm.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	Packed 8-bit permute.
shuff.pbh	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	Packed 8-bit shuffle returning high result.
shuff.pbl	$R_{IDEST} = R_{SRC1}, I_{SRC2}$	Packed 8-bit shuffle returning low result.
shuff.phh	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Packed 16-bit shuffle returning high result.
shuff.phl	$R_{DEST} = R_{SRC1}, I_{SRC2}$	Packed 16-bit shuffle returning low result.
shuffeve.pb	$R_{DEST} = R_{SRC1}, I_{SRC2}$	Packed 8-bit shuffle of even fields.
shuffodd.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Packed 8-bit shuffle of odd fields.
Operations available as macros		
slctf.pb	$R_{IDEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	Byte select between input operands using negated 4-bit condition code in branch register.

Table 26. Packed operations summary table (Continued)

Operation	Syntax	Description
unpacku.pbh	$R_{DEST} = R_{SRC1}$	Unpack upper two 8-bit values into 16-bit results.
unpacku.pbl	$R_{DEST} = R_{SRC1}$	Unpack lower two 8-bit values into 16-bit results.
pack.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Pack lower 16-bit value from each operand into 32-bit result.

9 SIMD operations

The ST240 instruction set contains a large number of SIMD (Single Instruction, Multiple Data) operations. These typically perform a single operation on multiple packed data fields. In some cases packed results are written and in some cases a single result is written. SIMD operations were introduced in *Chapter 8: Logical operations* and *Chapter 7: Arithmetic operations* and are described in this chapter.

Some operations involve rounding as described in *Section 7.1.1: Rounding on page 36*. The operand types are described in *Section 7.1.2: Operand types on page 38*.

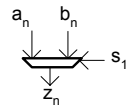
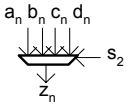
9.1 Notation used in this chapter

The diagrams in this chapter use a specific notation to indicate the steps involved in each operation.

Table 27. Notation used for SIMD diagrams

Notation used in diagrams	Description
$+_n(a,b)$, $-_n(a,b)$	Add or subtract n bit values a and b. The result has n+1 bits.
$+_n(a,b,c,d)$	Add 4 n bit values. The result has n+2 bits.
$Clip_n(a)$	Return the n least significant bits of a, discarding any remaining upper bits.
$Sat_n(a)$	$Saturate_n(a)$
$Abs_n(a)$	Return the absolute value of the n bit value a. The result has n+1 bits.
$Xs_n(a,b)$	Perform a signed multiplication of two n bit numbers a and b. The result has 2n bits.
$Xus_n(a,b)$	Perform an unsigned (value a) by signed (value b) of two n bit numbers. The result has 2n bits.
$RM_n(a)$, $RNE_n(a)$,$RNP_n(a)$	Round the upper n bits of a using the given rounding mode.
$<<_n(a,b)$	Shift the n bit number a left b places. The result has n + b bits.
$Min_n(a,b)$, $Max_n(a,b)$	Return the minimum or maximum of the n bit numbers a and b. The result has n bits.
$USat_n(a)$, $Sat_n(a)$	Unsigned $Saturate_n(a)$, $Saturate_n(a)$ as defined in *Chapter 23*.
$=_n$	n bit equality comparison. The result is 1-bit boolean.
$/2_n(a)$	Integer division of n bit value a by 2. The result has n-1 bits.
$/4_n(a)$	Integer division of n bit value a by 4. The result has n-2 bits.

Table 27. Notation used for SIMD diagrams (Continued)

Notation used in diagrams	Description
	Two input multiplexer. Data inputs a and b have n bits and the selection input s has 1 bit. When s = 0, a is selected by the output z, and when s = 1, b is selected.
	Four input multiplexer. Data inputs a to d have n bits and the selection input s has 2 bits. When s = 00, a is selected by the output z, and when s = 01, b is selected and so on.

9.2 SIMD 16-bit arithmetic operations

SIMD 16-bit arithmetic operations are performed on signed values. These operations are summarized in *Table 28*.

Table 28. SIMD 16-bit arithmetic operations

Operation	Syntax	Latency	Description
abss.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed absolute with saturation. Operands may be signed 16-bit integers or fractional 1.15 format.
add.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit addition. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
adds.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed addition with saturation. Operands may be signed 16-bit integers or fractional 1.15 format.
cmpeq.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit test for equality
cmpgt.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed greater than
max.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed maximum. Operands may be signed 16-bit integers or fractional 1.15 format.
min.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed minimum. Operands may be signed 16-bit integers or fractional 1.15 format.
mul.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit multiplication. Operands may be signed or unsigned 16-bit integers.
muladd.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit signed multiplication and add across with saturation. Operands are signed 16-bit integers.

Table 28. SIMD 16-bit arithmetic operations (Continued)

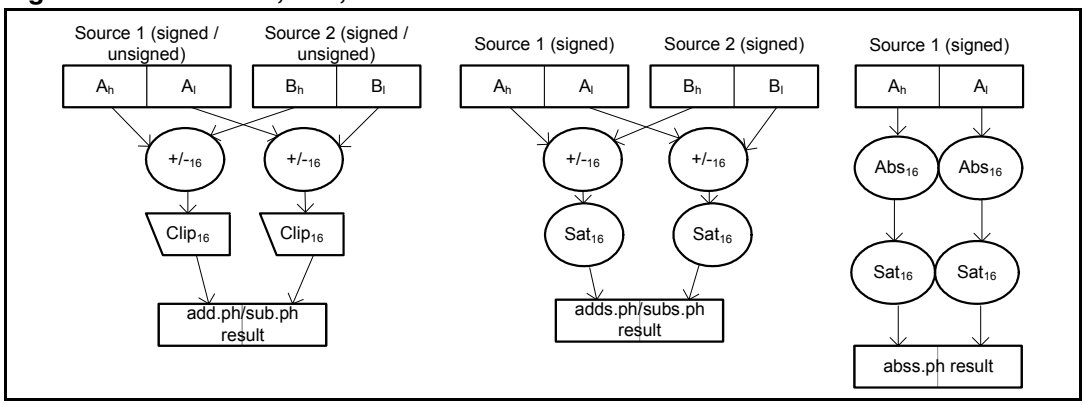
Operation	Syntax	Latency	Description
mulfracadds.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit signed fractional multiplication and add across with saturation. Operands are fractional 1.15 fractional format.
mulfracrm.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit fractional multiplication with round minus and saturation. Operands are fractional 1.15 format.
mulfracrne.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Packed 16-bit fractional multiplication with round nearest even and saturation. Operands are fractional 1.15 format.
shl.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Packed 16-bit left shift. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
shls.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift left with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
shr.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	1	Packed 16-bit arithmetic right shift. Operands may be signed 16-bit integer or fractional 1.15 format.
shrrne.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift right with round nearest even rounding. Operands may be signed 16-bit integer or fractional 1.15 format.
shrrnp.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	2	Packed 16-bit signed shift right with round nearest positive rounding. Operands may be signed 16-bit integer or fractional 1.15 format.
sub.ph	$R_{DEST} = R_{SRC2}, R_{SRC1}$	1	Packed 16-bit subtraction. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.
subs.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 16-bit signed subtraction with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

9.2.1 SIMD 16-bit add and subtract operations

The packed 16-bit addition and subtraction operations, **add.ph** and **sub.ph,** are supported with and without saturation. When saturation is applied all operands are treated as signed, without saturation the operands maybe signed or unsigned. Signed operands can be integers or 1.15 fractional format.

These operations are illustrated in *Figure 5*.

Figure 5. SIMD add, sub, absolute



add.ph and **sub.ph** can internally overflow, as represented by the $Clip_{16}()$ function. To avoid this the range of operands should be restricted or the saturating versions used.

A signed absolute with saturation is also defined (**abss.ph**). This includes saturation for the case where any field of the input operand has the negative maximum value.

9.2.2 SIMD 16-bit multiplication operations

The **mul.ph** operation multiplies pairs of 16-bit values and returns a 16-bit result in each half of the result word. Because the upper bits of the result are discarded either signed or unsigned operands may be used. If the upper bits of the result are required, the scalar operations **mulll** or **mulllu** must be used.

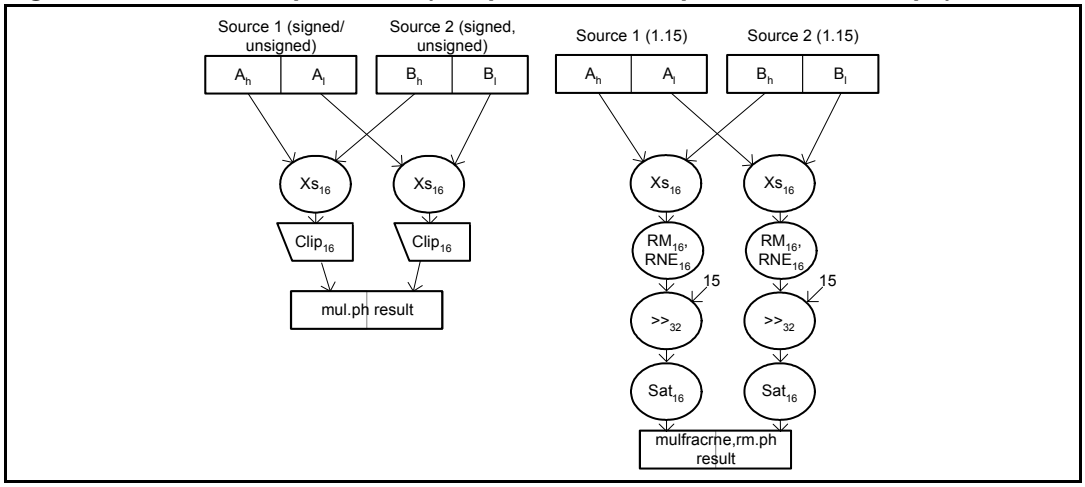
The packed 16-bit fractional multiply operations **mulfracrne.ph** and **mulfracrm.ph** treat their operands as fractional values in the format 1.15. The results are in the same format, but are rounded differently (RM and RNE rounding modes).

In both cases the result is saturated, but this is only needed for -1.0 multiplied by -1.0 as the result, +1.0, is outside the bounds of a 1.15 fraction.

mulfracrm.ph is used for an ETSI function as described in *Table 7.6.2: Saturating operations usage for implementation of ETSI functions on page 47*.

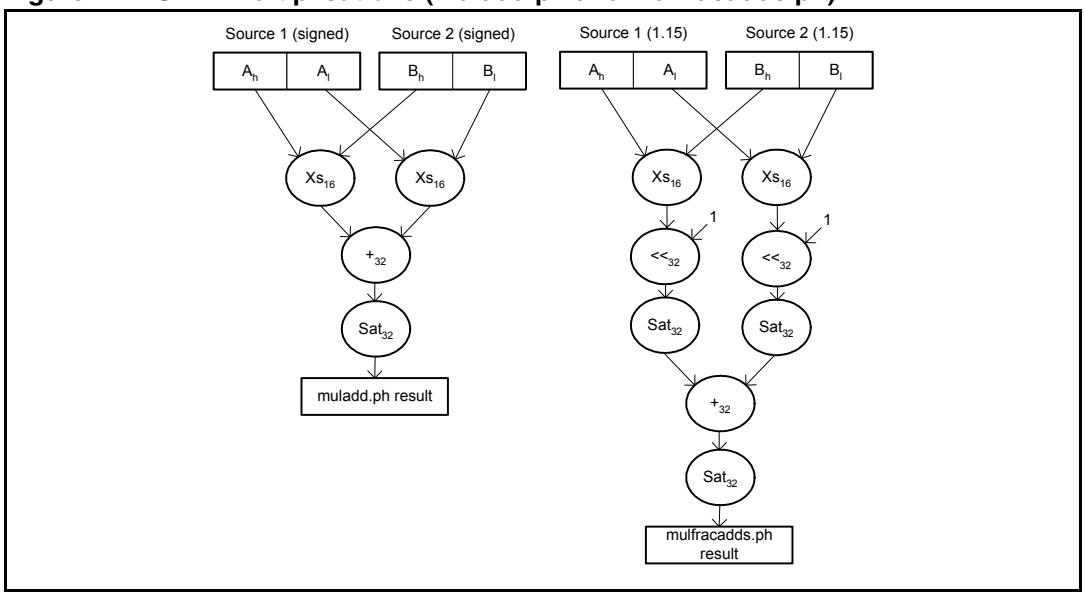
The SIMD multiplications **mul.ph, mulfracrne.ph** and **mulfracrm.ph** are illustrated in *Figure 6*.

Figure 6. SIMD multiplications (mul.ph, mulfracrne.ph and mulfracrm.ph)



The **muladd.ph** operation performs two simultaneous signed 16-bit multiplies accumulating the two 32-bit intermediates into a single 32-bit result. The only loss in accuracy for this operation is when all four source elements are the most negative 16-bit integer. The **mulfracadds.ph** performs two simultaneous 1.15 fractional multiplies, converts the 2.30 results to 1.31 format (using a one-place saturating left shift) then adds the two results together using a saturating add. The operation is designed to implement ETSI functions. The **muladd.ph** and **mulfracadds.ph** operations are illustrated in *Figure 7*.

Figure 7. SIMD multiplications (muladd.ph and mulfracadds.ph)

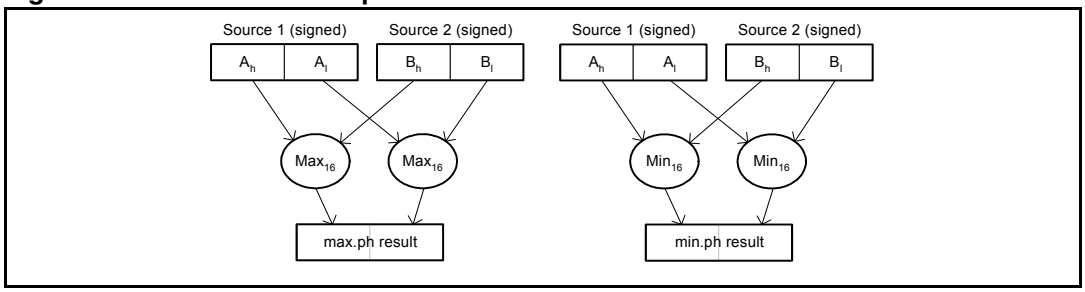


9.2.3 SIMD 16-bit comparison operations

The comparison operations, **cmpeq.ph** and **cmpgt.ph**, return 0 or 1 in the least significant bit of each field of the result. If the destination is a branch register, the destination fields are bits [3:2] and [1:0]. If the destination is a general purpose register, the destination fields are bits [31:16] and [15:0]. For branch registers, the result is propagated to all bits in the field. For general purpose registers, bits other than the result are set to 0.

The maximum and minimum functions, **max.ph** and **min.ph**, are SIMD signed 16-bit operators that are applied independently to each field of the computation. These are probably the most useful of the operators involving comparison and can be used, for example, to clamp or saturate to arbitrary bounds and in median filtering. These operations are illustrated in *Figure 8*.

Figure 8. SIMD min/max operations

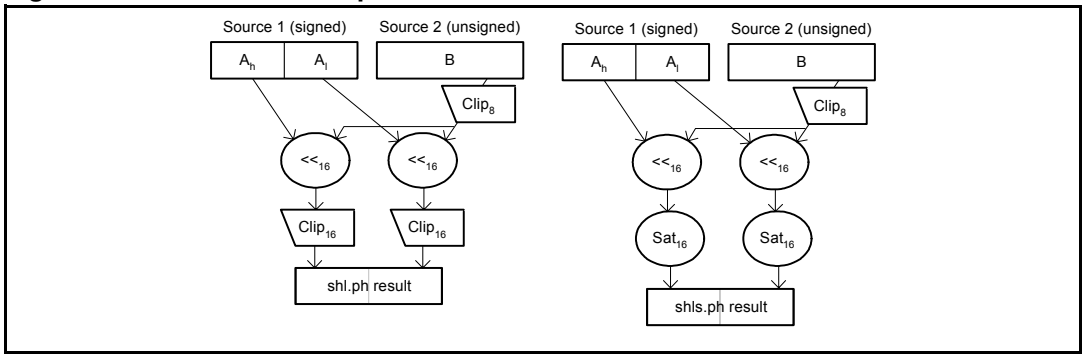


9.2.4 SIMD 16-bit shift operations

In shift operations the second operand is not packed and indicates the size of the shift to be applied to all fields of the operation. Consistently with the scalar shift operations only the first 8-bits of the shift operand are significant and are interpreted as an unsigned 8-bit value.

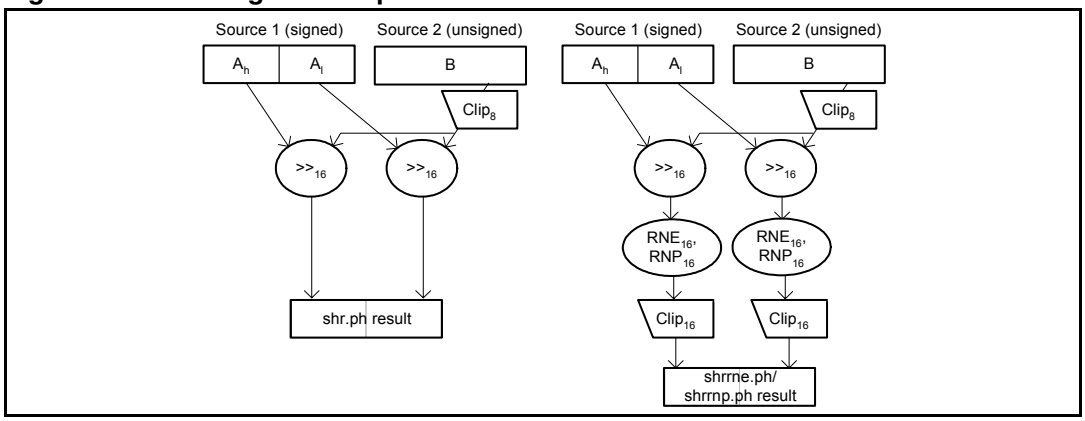
The left shift operations, **shl.ph** and **shls.ph**, are illustrated in *Figure 9*. The figure illustrates the difference between **shl.ph**, which clips the intermediate result to 16 bits and **shls.ph**, which saturates the intermediate result to 16 bits.

Figure 9. SIMD left shift operations



The right shift operations **shr.ph**, **shrrne.ph** and **shrrnp.ph** are signed. These operations are illustrated in *Figure 10*. **shr.ph** has no rounding applied, which is equivalent to round minus (RM) mode. **shrrne.ph** and **shrrnp.ph** have the rounding modes indicated by the opcode.

Figure 10. SIMD right shift operations



9.3 SIMD 8-bit arithmetic operations

SIMD 8-bit operations support the efficient implementation of functions that occur frequently in video and imaging applications. The operations described in this section treat their source operands as vectors of unsigned bytes. **muladdus.pb** is an exception where the second operand contains signed bytes. The reason for this is that the low-pass filter coefficients applied during MPEG video motion compensation and image processing are typically small signed values. The full set of 8-bit SIMD operations are listed in *Table 29*.

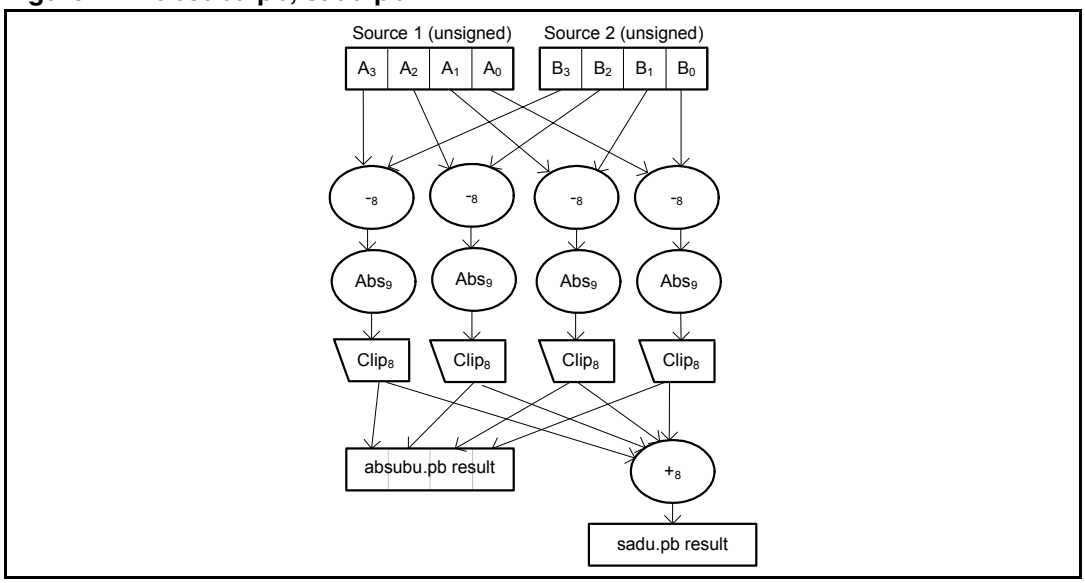
Table 29. SIMD 8-bit arithmetic operations

Operation	Syntax	Latency	Description
absubu.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	1	Packed 8-bit unsigned absolute difference.
avgu.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	1	Packed 8-bit unsigned average with selectable rounding mode (round zero or round positive).
avg4u.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	2	Packed 8-bit unsigned 4-way average with selectable rounding mode (round zero, round nearest negative, round nearest positive or round positive).
cmpeq.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 8-bit test for equality.
cmpgtu.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $B_{DEST2} = R_{SRC1}, R_{SRC2}$	1	Packed 8-bit unsigned greater than
muladdus.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	3	Unsigned 8-bit integer multiplied by signed 8-bit integer value and add across.
sadu.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	2	Sum of absolute differences on packed unsigned 8-bit values.

9.3.1 SIMD 8-bit absolute difference operations

Two absolute difference operations, **absubu.pb** and **sadu.pb**, are described. The first performs the operation **abs(a-b)** on each byte field of the source operands (treating each byte as an unsigned value) and returns the result in the corresponding byte field of the destination register. The second performs the same operation on the individual byte fields, but then sums these four values and returns the result. The two operations are illustrated in *Figure 11*.

Figure 11. absubu.pb, sadu.pb



absubu.pb is useful in image and video processing, for example, to implement the threshold checks performed by the deblocking filter built into an H264 decoder. The key application for **sadu.pb** is to implement the costing function in an MPEG encoder's motion estimator.

9.3.2 SIMD 8-bit averaging operations

Two averaging operations, **avgu.pb** and **avg4u.pb**, are defined, both take an additional source operand: a branch register that can be used to apply the required rounding. The same rounding mode is applied to all fields of the computation.

Two-way averaging performs the operation **(a + b + (c<>0))/2** four times, once on each byte field. Here **a** and **b** are from the same byte field index in the two 32-bit register sources and **c** is the contents of the branch register. The result is returned in the corresponding byte field of the destination register.

Four-way averaging performs the operation **(a + b + c + d + (e^3))/4** twice, one is performed on the lower two bytes from the two 32-bit source operands, the other is performed on the upper two bytes. Here **e** is the contents of the branch register.

The final term in the addition controls the rounding mode of the operation:

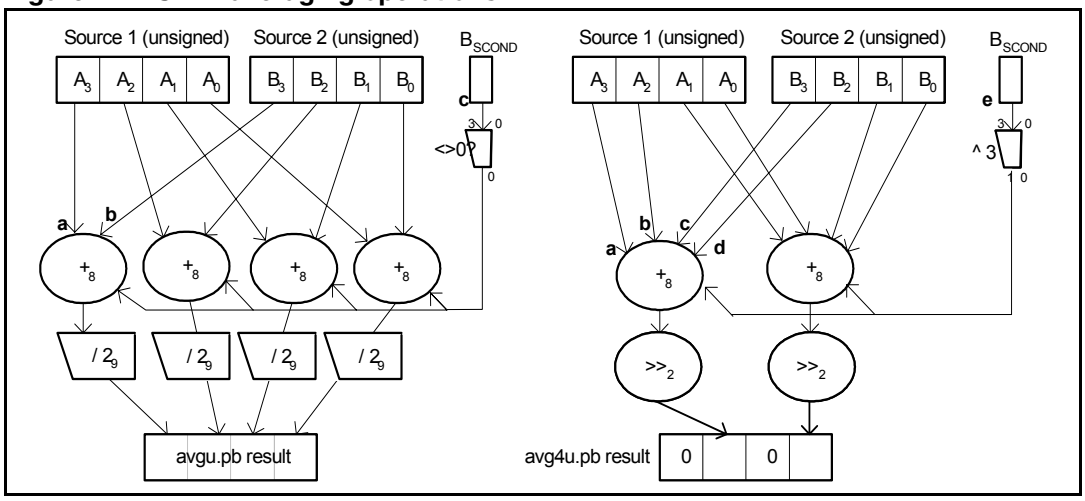
- **avgu.pb**: when c is zero the rounding mode is RZ, otherwise RP
- **avg4u.pb:** selects between four rounding modes as shown in *Table 30*

Table 30. avg4u.pb rounding mode selection

e ^ 3	Rounding
0	RZ
1	RNN
2	RNP
3	RP

Both operations are illustrated in *Figure 12*.

Figure 12. SIMD averaging operations



The two way average is useful for performing horizontal (when combined with **ext1.pb**) or vertical half-pel interpolations. The four-way average is useful when simultaneous horizontal and vertical interpolation is required. Using the branch register allows rounding to be parameterized by the interpolation function without causing much code replication.

As an illustration the following code fragment implements a vertical and horizontal interpolation of the 4 x 4 array in registers **a0**, **a1**, **a2** and **a3**. Register **a4** contains the next row of pixels, registers **b0** to **b4** contain the next column of pixels and the results are written to registers **c0** to **c4**.

```
avg4u.pb c0l = rnd, a0, a1
avg4u.pb c1l = rnd, a1, a2
avg4u.pb c2l = rnd, a2, a3
avg4u.pb c3l = rnd, a3, a4

ext1.pb  ab0 = a0, b0
ext1.pb  ab1 = a1, b1
ext1.pb  ab2 = a2, b2
ext1.pb  ab3 = a3, b3

ext1.pb  ab4 = a4, b4
avg4u.pb c0h = rnd, ab0, ab1
avg4u.pb c1h = rnd, ab1, ab2
avg4u.pb c2h = rnd, ab2, ab3

avg4u.pb c3h = rnd, ab3, ab4
shuffeve.pb c0 = c0l, c0h
shuffeve.pb c1 = c1l, c1h
shuffeve.pb c2 = c2l, c2h

shuffeve.pb c3 = c3l, c3h
; plus 3 spare slots
```

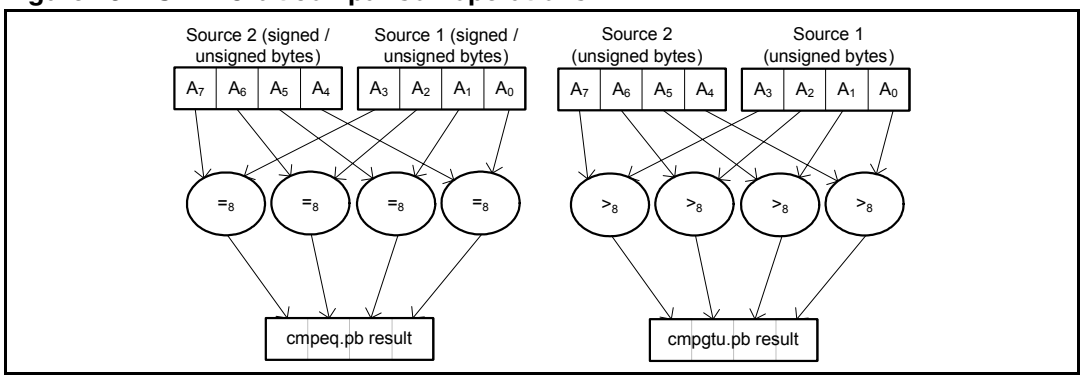
This code uses 17 syllables and will execute in 5 cycles, which compares with 64 syllables to implement in scalar code.

9.3.3 SIMD 8-bit comparison operations

The comparison operations, **cmpeq.pb** and **cmpgtu.pb**, return 0 in the least significant bit of the result field when false and 1 when true. For branch register destinations each result field is one bit of the result. For general purpose register destinations each result field is one byte. Bits other than the result are set to 0

The behaviors of **cmpeq.pb** and **cmpgtu.pb** are illustrated in *Figure 13*.

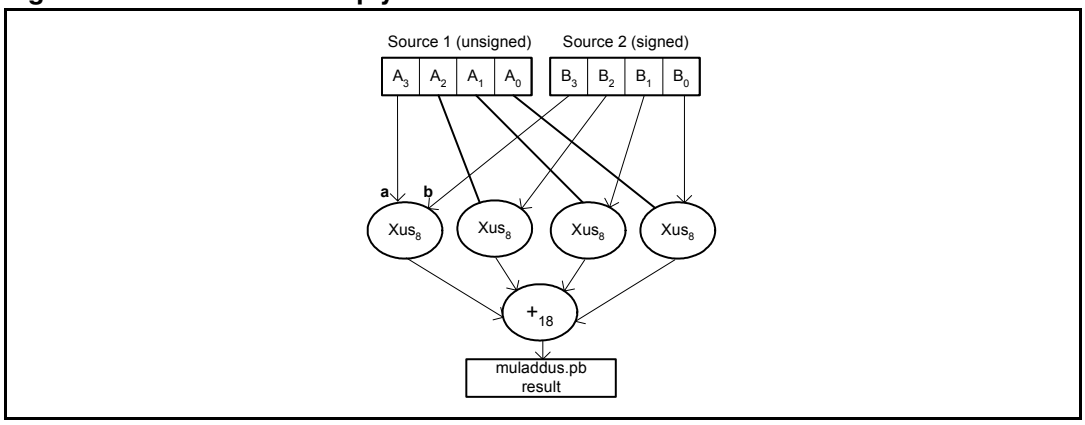
Figure 13. SIMD 8-bit comparison operations



9.3.4 SIMD 8-bit multiply and add across operation

The **muladdus.pb** operation multiplies an unsigned byte by a signed byte in each of the byte fields and then sums across the four fields to produce a single result. The operation is illustrated in *Figure 14*.

Figure 14. SIMD 8-bit multiply and add across



This operation is useful in video and image processing for applying low-pass filters to pixel vectors, for example, in quarter-pel interpolation and deblocking functions.

9.4 SIMD data manipulation operations

These operations are used to prepare data to ensure that the elements upon which the SIMD arithmetic operations are performed are aligned. Those operations defined for the ST240 SIMD instruction set are listed in *Table 31*.

Table 31. SIMD logical operations summary table

Operation	Syntax	Description
Simple logical primitives		
slct.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$ $R_{IDEST} = B_{SCOND}, R_{SRC1}, I_{SRC2}$	Byte select between input operands using 4-bit condition code in branch register.
slctf.pb	$R_{IDEST} = B_{SCOND}, R_{SRC1}, I_{SRC2}$	Byte select between input operands using negated 4-bit condition code in branch register.
Extraction operations		
ext1.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Extract word starting from byte 1.
ext2.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Extract word starting from byte 2.
ext3.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Extract word starting from byte 3.
extl.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	Dynamic extract left operation.
extr.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	Dynamic extract right operation.
Packing operations		
pack.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Pack four 16-bit values to 8-bit results ignoring upper bits.

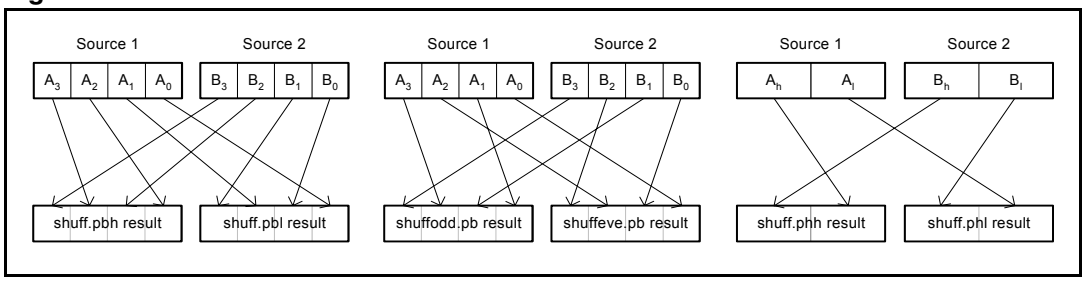
Table 31. SIMD logical operations summary table (Continued)

Operation	Syntax	Description
packrnp.phh	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Pack high part of 32-bit signed value into 16-bit signed results with round nearest positive.
packs.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Pack 32-bit signed values into 16-bit signed results with saturation.
packsu.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Pack 16-bit signed values into 8-bit unsigned results with saturation.
Shuffle and permute operations		
perm.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	Packed 8-bit permute
shuff.pbh	$R_{DEST} = R_{SRC1}, R_{SRC2}$ $R_{DEST} = R_{SRC1}, I_{SRC2}$	Packed 8-bit shuffle returning high result
shuff.pbl	$R_{IDEST} = R_{SRC1}, I_{SRC2}$	Packed 8-bit shuffle returning low result
shuff.phh	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Packed 16-bit shuffle returning high result
shuff.phl	$R_{DEST} = R_{SRC1}, I_{SRC2}$	Packed 16-bit shuffle returning low result
shuffeve.pb	$R_{DEST} = R_{SRC1}, I_{SRC2}$	Packed 8-bit shuffle of even fields
shuffodd.pb	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Packed 8-bit shuffle of odd fields
Operations available as macros		
slctf.pb	$R_{IDEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	Byte select between input operands using negated 4-bit condition code in branch register.
unpacku.pbh	$R_{DEST} = R_{SRC1}$	Unpack upper two 8-bit values into 16-bit results
unpacku.pbl	$R_{DEST} = R_{SRC1}$	Unpack lower two 8-bit values into 16-bit results
pack.ph	$R_{DEST} = R_{SRC1}, R_{SRC2}$	Pack lower 16-bit value from each operand into 32-bit result

9.4.1 SIMD shuffle operations

The shuffle operations (see *Table 31 on page 74*) provide a set of interleaving operations that can be applied to 8-bit and 16-bit packed data. They can be used for conversions between different packed data formats, transposing matrices, aligning data for butterfly operations and so on. These operations are illustrated in *Figure 15*.

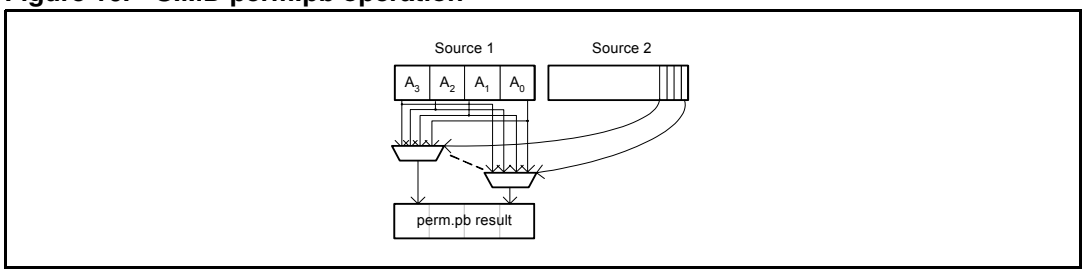
Figure 15. SIMD shuffles



9.4.2 SIMD permute operation

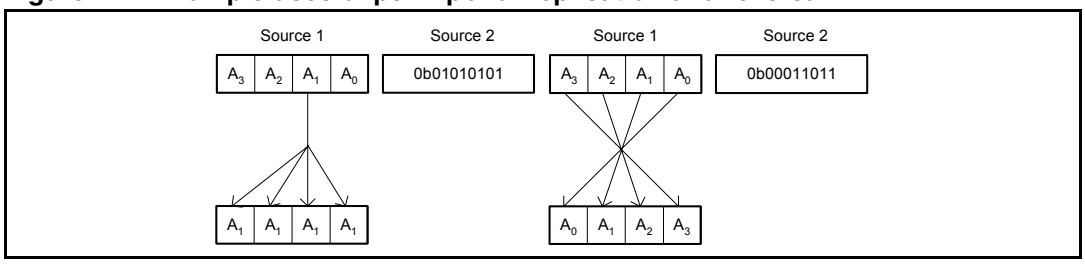
The **perm.pb** operation performs byte permutations. The two least significant bits of the second operand determine which byte from the first source operand is copied into the least significant byte of the result, shown as arrows controlling the muxes in *Figure 16*. The next two bits of the control (bits [3:2]) select the value for the next most significant byte of the result and so on for all four bytes of the result. The **perm.pb** operation is illustrated in *Figure 16*.

Figure 16. SIMD perm.pb operation



This operation can be used to replicate 8-bit and 16-bit fields throughout a packed vector. It can also be used to reverse the order of 16-bit or 8-bit fields. Many other permutations are possible. Examples of this operation are illustrated in *Figure 17*.

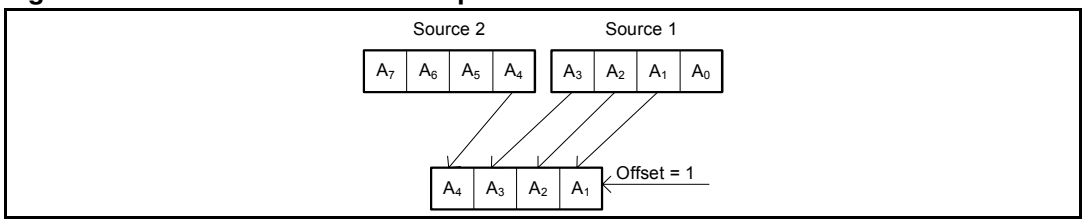
Figure 17. Example uses of perm.pb for replication and reversal



9.4.3 SIMD static extraction operations

The extraction operations, **ext1.pb**, **ext2.pb** and **ext3.pb**, concatenate two 32-bit source operands to form a 64-bit intermediate result then extract a contiguous 32-bit sub-vector from this at a specified 8-bit offset. The operation is illustrated in *Figure 18*.

Figure 18. SIMD static extraction operations



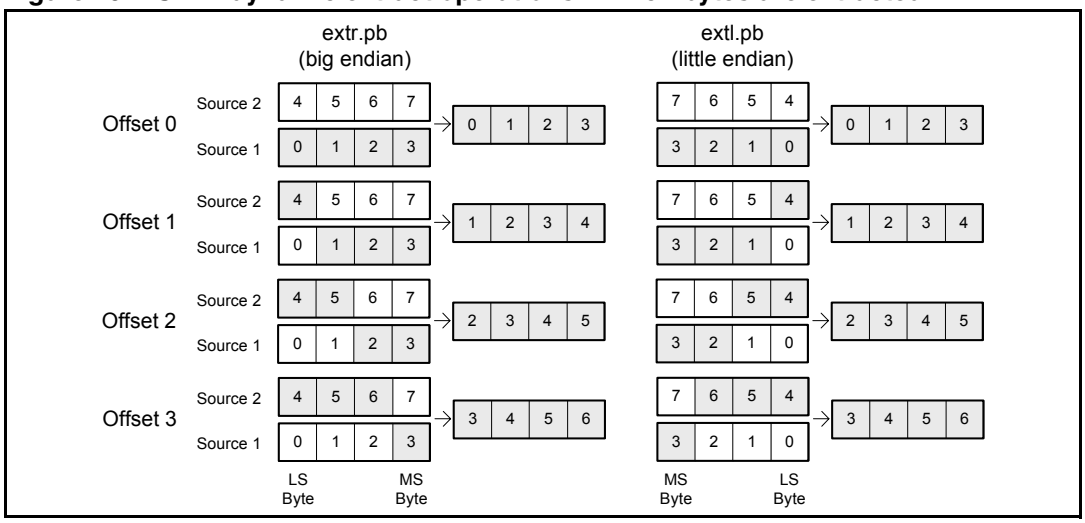
SIMD extraction is effectively a four operand operation: two register sources, an immediate specifying the offset and the result register. Four operand operations are not possible, so extract is split into three different operations, with the possible offsets of [1, 3] represented in the opcode.

Extractions are useful for aligning adjacent data elements ready for a vertical operation, for example, the byte averaging performed during horizontal half-pixel interpolation.

9.4.4 SIMD dynamic extraction operations

Dynamic extraction operations, **extl.pb** and **extr.pb**, use the branch registers as a third source operand for the extraction operations, so that the offset can be dynamic rather than static. Only the two least significant bits of the branch register are used to generate the offset, if the offset is zero, the result is the contents of the first operand. The operation is illustrated in *Figure 19* where *Offset* represents the two least significant bits of the branch register operand. **extr.pb** extracts bytes using big endian byte order and **extl.pb** extracts using little endian byte order. As is shown, the bytes are correctly ordered for each endianness when the correct operation is used.

Figure 19. SIMD dynamic extract operations - which bytes are extracted



These operations are useful for the handling of data misaligned with respect to the load/store operations used to access it. This is discussed further in *Section 9.4.7: Handling unaligned data using logical SIMD operations on page 79*.

9.4.5 SIMD pack operations

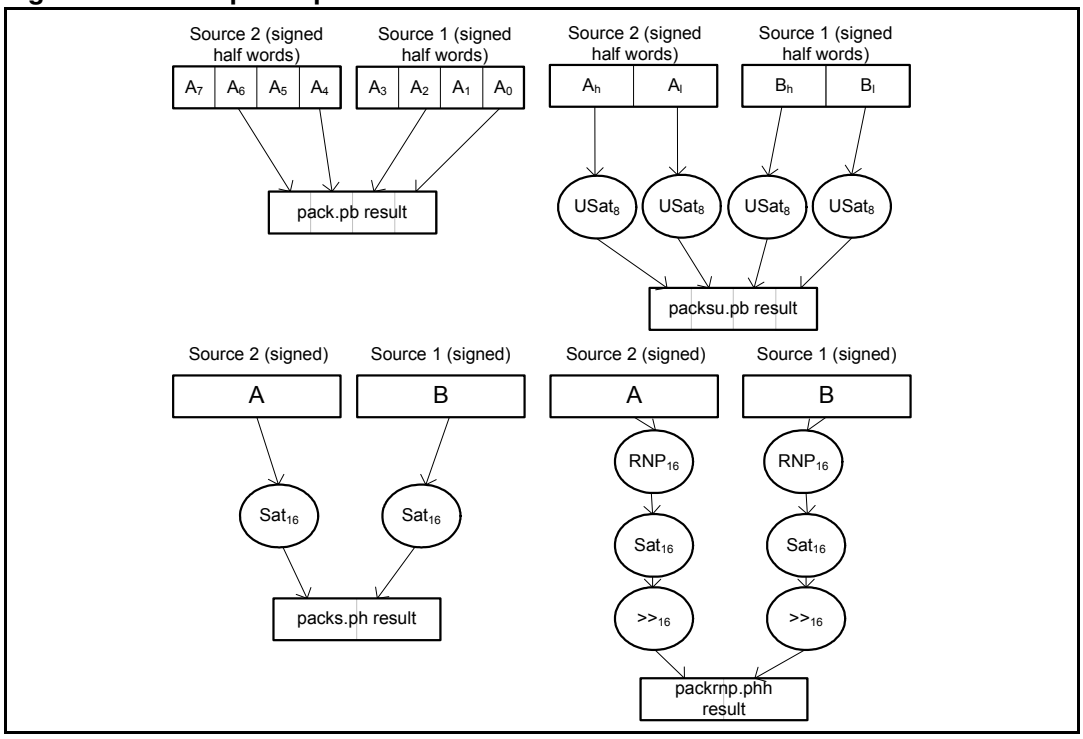
Four operations are provided for packing.

- **packs.ph**: signed 32-bit values by saturating them to a signed 16-bit range.
- **packrnp.phh**: signed 32-bit values by performing a rounding signed right shift down to a 16-bit range.

  This operation includes the saturation necessary to handle input values close to the most positive integer.
- **pack.pb**: 16-bit values by ignoring the upper byte and packing the lower bytes together.
- **packsu.pb**: signed 16-bit values by saturating them to an unsigned 8-bit range.

These operations are illustrated in *Figure 20*.

Figure 20. SIMD pack operations

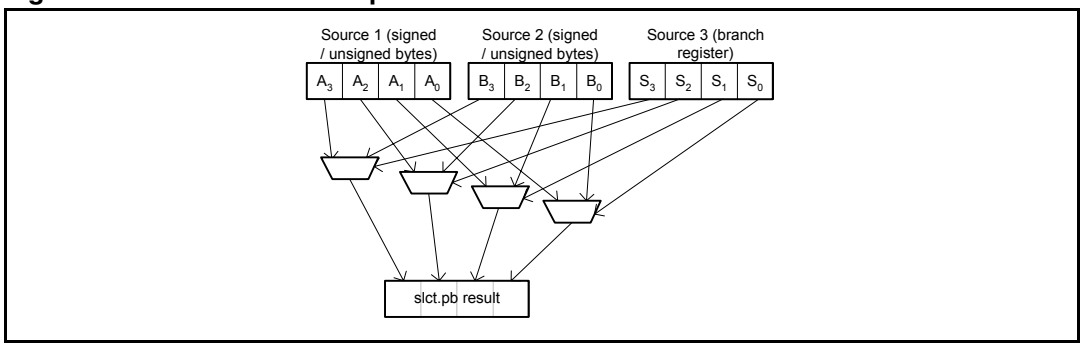


Note: ***pack.ph*** *is implemented as a macro for the* ***shuff.phl*** *operation.*

9.4.6 SIMD selection operations

The select operations, **slct.pb** and **slctf.pb,** choose between the values in the source operand byte fields based on the setting of the bit in the corresponding field of the branch register. If the relevant bit of the branch register is true in a **slct.pb** operation the byte is taken from the first register operand, when false it is taken from the second, **slctf.pb** selects the other byte. Only byte forms are needed because the 16-bit branch compares set both bits of the corresponding field in the branch register. This operation is illustrated in *Figure 21*.

Figure 21. SIMD selection operations



A **slctf.pb** where the second operand is a register is implemented as a macro that swaps the operands to **slct.pb**.

9.4.7 Handling unaligned data using logical SIMD operations

The dynamic extract operations, **extl.pb** and **extr.pb**, are useful for accessing data that is misaligned with respect to the width of the load/store operations used to access it. The address alignment must be placed in a branch register using a **mov** operation.

Unaligned loads

When synthesizing unaligned loads in a little-endian system **extl.pb** gives the correct alignment behavior, but for big-endian systems **extr.pb** is required.

As an illustration of how the operations are used the following code fragment loads a 16-element byte array at an arbitrarily aligned address **addr** into registers R1 to R5.

```
mov  offset = addr
and    addr = addr, -4
;; ##+ 2 spare slots
ldw    $r1 = 0[addr]
;; ##+ 3 spare slots
ldw    $r2 = 4[addr]
;; ##+ 3 spare slots
ldw    $r3 = 8[addr]
;; ##+ 3 spare slots
ldw    $r4 = 12[addr]
;; ##+ 3 spare slots
ldw    $r5 = 16[addr]
extl.pb a0 = offset, u0, u1
;; ##+ 2 spare slots
extl.pb a1 = offset, u1, u2
;; ##+ 3 spare slots
extl.pb a2 = offset, u2, u3
;; ##+ 3 spare slots
extl.pb a3 = offset, u3, u4
;; ##+ 3 spare slots
```

In principal dynamic extraction operations can be combined with 64-bit loads. The issue is that the unaligned array could start in either the upper or lower 32-bit word and the extract sequence needs to begin with the starting word. This can be resolved either by using a branch or, probably more efficiently, a selection operation before each dynamic extraction operation.

Unaligned stores

Dynamic extraction operations can also be used to synthesize stores to arrays of unaligned values, some additional pre-amble and post-amble is required to handle the words only partially written to. By way of illustration, the following code fragment copies an integer array to an arbitrarily aligned address on a little-endian memory system.

```
typedef int __attribute__((__may_alias__)) simd_int;(1)

void simd_misalign_store(
  unsigned char *dst, unsigned int src[], int n)
{
   simd_int *dptr = (simd_int*)((unsigned int)dst & -4);
   unsigned int align = (unsigned int)dst;
   simd_int start = *dptr;(2)
   simd_int buffer = __st200extl_pb(align,0,start);(3)
   simd_int end;
  int i;
  for (i=0; i<n ; i++)
  {
    unsigned int val = src[i];
    dptr[i] = __st200extr_pb(align,val,buffer);
    buffer = val;
  }

  end    = dptr[n];
  end    = __st200extl_pb(align,end,0);
  dptr[n] = __st200extr_pb(align,end,buffer);

}
```

1. Creating a type with the attribute **`may_alias`** stops the compiler assuming the **`dst`** and **`dptr`** arrays are independent.
2. Need to load first aligned word of destination so it can be merged with first store.
3. This extract places the preserved bytes at the bottom of the alignment buffer.

9.5 Summary of SIMD branch register operations

A small set of operations is used to access branch registers. These are listed in *Table 32*.

Table 32. SIMD logical operations summary table

Operation	Syntax	Description
Selection operations		
slct.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$ $R_{IDEST} = B_{SCOND}, R_{SRC1}, I_{SRC2}$	Byte select between input operands using 4-bit condition code in branch register.
slctf.pb	$R_{IDEST} = B_{SCOND}, R_{SRC1}, I_{SRC2}$	Byte select between input operands using negated 4-bit condition code in branch register.
Extraction operations		
extl.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	Dynamic extract left operation.
extr.pb	$R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	Dynamic extract right operation.
Comparison operations		
cmpeq.pb	$B_{DEST} = R_{SRC1}, R_{SRC2}$	Packed 8-bit test for equality.
cmpeq.ph	$B_{DEST} = R_{SRC1}, R_{SRC2}$	Packed 16-bit test for equality.
cmpgtu.pb	$B_{DEST} = R_{SRC1}, R_{SRC2}$	Packed 8-bit unsigned greater than.
cmpgt.ph	$B_{DEST} = R_{SRC1}, R_{SRC2}$	Packed 16-bit greater than.
Operations available as macros		
slctf.pb	$R_{IDEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}$	Byte select between input operands using negated 4-bit condition code in branch register.

10 Traps (exceptions and interrupts)

In the ST240 architecture, exceptions and interrupts are jointly termed traps. The flow diagram, *Chapter 22: Execution model on page 199* shows the behaviour of the ST240 when a trap is taken. The aim of this chapter is to describe the types of traps which may be taken and to define the steps that are carried out when a trap is taken.

The ST240 defines two types of traps.

- *Normal traps* caused by program execution or interrupts.
- *Debug traps* caused by an external debugger requiring entry into debug mode.

10.1 Trap types

The sections that follow provide details of the different types of traps. All possible traps along with the associated trap types are listed in *Table 34: Trap types and priorities on page 84*.

10.1.1 Interrupt types

There are two types of interrupts: normal interrupts and debug interrupts.

Normal interrupts

Possible sources of normal interrupts are:

- external interrupt inputs
- timer generated interrupts
- performance monitor generated interrupts

More information about normal interrupts can be found in *Chapter 19: Interrupt controller on page 163*.

Debug interrupts

Debug interrupts can be raised from the following sources:

- the JTAG port under control of an external host
- a write to the RAISE_DEBUG_INT bit of DSR1
- the assertion of the trigger_in input pin

More information about debug interrupts can be found in *Chapter 20: Debugging support on page 169*.

10.1.2 Exception types

Both types of exceptions include breakpoints. More information about breakpoints can be found in *Chapter 20: Debugging support on page 169*.

Normal exceptions

Normal exceptions arise from program execution and from normal breakpoints. All exceptions that are not debug exceptions fit into this category.

Debug exceptions

Debug exceptions arise when debug breakpoints trigger. Debug breakpoints cause the ST240 to enter debug mode.

10.2 Non recoverable exceptions

A class of exceptions are non-recoverable; program execution cannot continue after the exception has been raised. Non-recoverable exceptions are described in *Chapter 22: Execution model on page 199*.

10.3 Trap mechanism

The trap point is the point in the program execution where a trap occurs. All bundles executed before the trap point have finished updating the architectural state; no architectural state has been updated by subsequent bundles. For an exception the trap point is the start of the bundle that caused the exception. For an interrupt, the trap point is the start of the bundle that had its execution interrupted. Typically the interrupted bundle had begun but not completed execution when the interrupt was raised or enabled.

In effect taking a trap can be viewed as executing an operation that branches to the required handler. The branch has a number of side effects which are described in this chapter.

At the trap point the ST240 transfers execution to the trap handler and saves the execution state as detailed in *Section 10.7: Saving and restoring execution state on page 87*. All operations issued before the trapping bundle are allowed to complete. All operations issued after (including the trapping bundle) are discarded. The architectural state, with the exception of saved execution state, remains as it is at the trap point. Hence ST240 interrupts and exceptions are precise.

10.4 Trap handling

The execution of a single bundle can result in multiple traps. The traps are prioritized as shown in *Table 34 on page 84* and only the highest priority trap is raised. Multiple traps resulting from the execution of a single bundle do not affect the execution of other bundles.

10.5 Trap vector and priorities

When a trap is taken, a handler address is chosen from the trap vector depending upon the trap type as shown in *Table 33*. Note that all normal traps cause the ST240 to jump to a programmable address set by writing to the control register listed in the table. Debug traps cause the ST240 to jump to the default debug handler; this is code contained within the DSU. This handler is referred to as DEBUG_HANDLER in this chapter for convenience even though this does *not* refer to an actual control register. See *Section 20.4.1: Default debug handler on page 181*.

Table 33. Trap vector

Handler	Type	Description
TRAP_TLB	Control register	This handler is used for TLB faults.
TRAP_BREAK		This handler is used for normal breakpoints.
TRAP_EXCEPTION		This handler is used for normal exceptions not covered by TRAP_TLB, TRAP_BREAK.
TRAP_INTERRUPT		This handler is used for normal interrupts.
DEBUG_HANDLER	Fixed address in DSU	This handler is used for all debug traps.

10.6 Trap priorities

Table 34 shows all possible traps with the handler address that each trap uses. The entries in the table are listed in decending priority order starting with the highest priority.

Table 34. Trap types and priorities

Priority	Name	Description	normal, debug	DBG_EXCAUSENO	EXCAUSENO	synchronous, asynchronous	Handler
1	EVENT	The host has requested a debug interrupt by sending an event message.	debug	0		async	DEBUG_ HANDLER
2	RAISE_DBG_INT	A debug interrupt has been caused by a write to DSR1.		1			
3	TRIGGER	The trigger_in pin has been asserted.		2			
4	STBUS_IC_ERROR	An STBus request caused by an instruction cache fill or a **prginsadd** or **prginsset** propagated to the L2 caused an STBus error.	normal		0	sync	TRAP_ EXCEPTION
5	STBUS_DC_ERROR(1)	An STBus request caused by a store operation or by a **prgadd**, **prgset**, **invadd** or **flushadd** propagated to the L2 caused an STBus error.(2)			1	async	
6	EXTERN_INT	An unmasked internal or external interrupt request was received.			2		TRAP_ INTERRUPT

Table 34. Trap types and priorities (Continued)

Priority	Name	Description	normal, debug	DBG_EXCAUSENO	EXCAUSENO	synchronous, asynchronous	Handler
7	DBG_IBREAK	A debug instruction address breakpoint triggered.	debug	3			DEBUG_ HANDLER
8	IBREAK	An normal instruction address breakpoint triggered.	normal		3		TRAP_ BREAK
9	ITLB	An instruction related TLB exception occurred.			4		TRAP_TLB
10	DBG_SBREAK	An enabled debug software breakpoint triggered.(3)	debug	4			DEBUG_ HANDLER
11	SBREAK	A normal software breakpoint triggered.	normal		5	synch	TRAP_BREAK
12	ILL_INST	The bundle could not be decoded into legal sequence of operations or a privileged operation is being issued in user mode.			6		TRAP_ EXCEPTION
13	SYSCALL	This exception is used as a hook to signal to the OS that user code requires a system call.			7		
14	DBG_DBREAK	A debug breakpoint on a data address triggered.	debug	5			DEBUG_ HANDLER

Table 34. Trap types and priorities (Continued)

Priority	Name	Description	normal, debug	DBG_EXCAUSENO	EXCAUSENO	synchronous, asynchronous	Handler
15	DBREAK	A normal breakpoint on a data address triggered.	normal		8		TRAP_ BREAK
16	MISALIGNED_TRAP	A load or store address is not aligned to the data width. Not thrown for accesses to control register space.			9		TRAP_ EXCEPTION
17	CREG_NO_MAPPING	A word load or store to control register space was attempted which didn't map to a valid control register.			10		TRAP_ EXCEPTION
18	CREG_ACCESS_ VIOLATION	A word load or store to a control register was attempted without the necessary privileges, or a byte, half-word or long word load or store to a control register was attempted.			11		
19	DTLB	A data related TLB exception occurred.			12		TRAP_TLB
20	DTCM_ONLY_ VIOLATION	DTCM_ONLY is set in the PSW and a load, store, prefetch, data cache purge or data cache invalidation operation is executed to a memory page which is 1. mapped as cached in the TLB 2.outside the DTCM address range.			13		TRAP_ EXCEPTION
21	SDI_TIMEOUT	One of the SDI interfaces timed out while being accessed			14		
22	STBUS_DC_ERROR(1)	An STBus request caused by a load or by a **sync**(2) propagated to the L2 caused a bus error.			1		

1. STBUS_DC_ERROR is split into two as loads will not be submitted to the STBus if any other trap has occurred on the bundle, therefore STBus errors from loads have the lowest priority. STBus errors resulting from stores arrive asynchronously and cannot be directly associated with the store operation which caused them.
2. The implementation may make an STBus error caused by a sync propagated to the L2 cache as priority 5 instead of 22 which means handling it asynchronously.
3. There are two cases which cause DBG_SBREAK to be thrown as an ILL_INST exception instead, see *Section 20.2.3: Software breakpoints on page 172*.

10.7 Saving and restoring execution state

Directly following a trap, the saved execution state defines the reason for the trap and the precise trap point in the execution flow of the ST240. Execution is resumed by executing an **rfi** (return from interrupt) operation. Two levels of nested trap are possible - a debug trap may interrupt the execution of a normal trap. For this reason the debug trap has an extra level of state saving for the current PC and PSW.

10.7.1 Normal trap startup behavior

Taking an normal trap can be summarized as:

```
// Branch to the exception handler
EXCAUSE ← HighestPriorityBitNormal();
PC ← NormalExceptionHandlerAddress(EXCAUSE);

// Store information for the handler
EXCAUSENO ← HighestPriorityNumberNormal();
EXADDRESS ← ExceptAddressNormal(EXCAUSE);
TLB_EXCAUSE ← TlbExcauseValue();

// Save the PSW and PC

SAVED_PSW ← PSW;
SAVED_PC ← BUNDLE_PC;

// clear the multi-processor/multi-threaded lock
ATOMIC_LOCK[LOCKED] ← 0;

// Enter supervisor mode.
// Disable interrupts, DTCM_ONLY mode, normal breakpoints

PSW[USER_MODE] ← 0;
PSW[INT_ENABLE] ← 0;
PSW[DTCM_ONLY] ← 0;
PSW[IBREAK_ENABLE] ← 0;
PSW[DBREAK_ENABLE] ← 0;
```

Function definitions

NormalExceptionHandlerAddress() returns the correct handler for the trap type as defined in *Table 34: Trap types and priorities on page 84*.

HighestPriorityNumberNormal() returns the number of the highest priority normal trap from those that have been raised as defined in *Table 34: Trap types and priorities on page 84* and in *Table 39: EXCAUSENO_EXCAUSENO values on page 90*.

HighestPriorityBitNormal() returns 1 << **HighestPriorityNumberNormal**(), see *Section 10.8.1: Normal traps on page 90*.

ExceptAddressNormal() defines the value that is stored into the EXADDRESS control register as shown in *Table 35*.

Table 35. ExceptAddressNormal() function definition

Trap type	Value
ITLB	The virtual address of the syllable which caused the ITLB exception - see *Section 11.5: EXADDRESS register or TLB exceptions on page 105*.
DBREAK	The virtual data address which caused the exception.
MISALIGNED_TRAP	
CREG_NO_MAPPING	
CREG_ACCESS_VIOLATION	
DTLB	
All other traps	Zero

Any result from **ExceptAddressNormal**() which is not defined to be zero is the optional argument that is passed to **THROW** (see *Section 23.3.5: Exceptions on page 212*) when the trap was raised.

TlbExcauseValue() defines the value that is stored into the TLB_EXCAUSE control register as shown in *Table 36*.

Table 36. TlbExcauseValue() function definition

Trap type	Value
ITLB	Information regarding the TLB exception which has been thrown - see *Section 11.4.11: TLB_EXCAUSE register on page 103*.
DTLB	
All other traps	Zero

10.7.2 Debug trap startup behavior

Taking an debug trap is summarized in the following pseudo code:

```
// Branch to the debug handler
PC ← DEBUG_HANDLER;
// Store information for the handler
DBG_EXCAUSENO ← HighestPriorityDebug();
DBG_EXADDRESS ← ExceptAddressDebug(DBG_EXCAUSENO);
// Save the PSW and PC
SAVED_PSW ← PSW;
SAVED_PC ← BUNDLE_PC;
SAVED_SAVED_PC ←SAVED_PC;
SAVED_SAVED_PSW ←SAVED_PSW;

// clear the multi-processor lock
ATOMIC_LOCK[LOCKED] ← 0;
// Enter debug (supervisor) state
// Disable interrupts, DTCM_ONLY mode, normal breakpoints, TLB
PSW[USER_MODE]  ← 0;
PSW[DEBUG_MODE] ← 1;
PSW[INT_ENABLE] ← 0;
PSW[DTCM_ONLY]   ← 0;
PSW[IBREAK_ENABLE] ← 0;
PSW[DBREAK_ENABLE] ← 0;
PSW[TLB_ENABLE] ← 0;
```

Function definitions

HighestPriorityDebug() returns the highest priority debug trap from those that have been raised as defined in *Table 34: Trap types and priorities on page 84* and in *Table 41: DBG_EXCAUSENO_DBG_EXCAUSENO values on page 92*.

ExceptAddressDebug() defines the value that is stored into the DBG_EXADDRESS control register as shown in *Table 37*.

Table 37. ExceptAddressDebug() function definition

Trap type	Value
DBG_DBREAK	The virtual data address that caused the exception.
All other debug traps	Zero

10.7.3 Restoring execution state

An **rfi** (return from interrupt) operation is used to restart execution. The **rfi** operation causes the following state updates:

```
PC ← SAVED_PC;
PSW ← SAVED_PSW;
SAVED_PC ← SAVED_SAVED_PC;
SAVED_PSW ← SAVED_SAVED_PSW;
ATOMIC_LOCK[LOCKED] ←0;
ATOMIC_LOCK[SHADOW_LOCKED] ←0;
```

The saved PC and PSW values may be modified after the trap has been taken and before executing the **rfi**. Therefore execution need not:

- return to the trapping bundle
- restore the PSW value which was active when the trap occurred

10.8 Determining the trap type

This section describes the causes of normal and debug traps.

10.8.1 Normal traps

The EXCAUSENO control register (*Table 38*) provides the cause of the last normal trap.

Table 38. EXCAUSENO bit fields

Name	Bit(s)	Writeable	Reset	Comment
EXCAUSENO	[4:0]	RW	0x0	Specifies the trap number.
Reserved	[31:5]	RO	0x0	Reserved.

For backward compatibility the exception cause is also available as a bit field by reading the EXCAUSE register. The EXCAUSE register is read only and always returns 1<<EXCAUSENO. The meaning of each value is given in *Table 39*, and the descriptions are given in *Table 34: Trap types and priorities on page 84*.

Table 39. EXCAUSENO_EXCAUSENO values

Name	Value	Comment
STBUS_IC_ERROR	0	An STBus request caused by an instruction cache fill caused a bus error.
STBUS_DC_ERROR	1	An STBus request caused by a load or store operation caused a bus error.
EXTERN_INT	2	An unmasked internal or external interrupt request was received.
IBREAK	3	A normal instruction address breakpoint triggered.
ITLB	4	An instruction related TLB exception occurred.
SBREAK	5	A normal software breakpoint triggered.

Table 39. EXCAUSENO_EXCAUSENO values (Continued)

Name	Value	Comment
ILL_INST	6	The bundle could not be decoded into legal sequence of operations or a privileged operation is being issued in user mode.
SYSCALL	7	This exception is used as a hook to signal to the OS that user code requires a system call.
DBREAK	8	A normal breakpoint on a data address triggered.
MISALIGNED_TRAP	9	A load or store address is not aligned to the data width. Not thrown for accesses to control register space.
CREG_NO_MAPPING	10	A word load or store to control register space was attempted which did not map to a valid control register.
CREG_ACCESS_VIOLATION	11	A word load or store to a control register was attempted without the necessary privileges, or a byte, half-word or long word load or store to a control register was attempted.
DTLB	12	A data related TLB exception occurred.
DTCM_ONLY_VIOLATION	13	DTCM_ONLY is set in the PSW and a load, store, prefetch, data cache purge or data cache invalidation operation is executed to a memory page which is 1.mapped as cached in the TLB 2.outside the DTCM address range.
SDI_TIMEOUT	14	One of the SDI interfaces timed out while being accessed.
Reserved	15-31	Reserved.

10.8.2 Debug traps

For a debug trap the cause is read from DBG_EXCAUSENO. The descriptions are given in *Table 34: Trap types and priorities on page 84*

Table 40. DBG_EXCAUSENO bit fields

Name	Bit(s)	Writeable	Reset	Comment
DBG_EXCAUSENO	[2:0]	RW	0x0	Specifies the debug trap number.
Reserved	[31:3]	RO	0x0	Reserved.

Table 41. DBG_EXCAUSENO_DBG_EXCAUSENO values

Name	Value	Comment
EVENT	0	The host has requested a debug interrupt by sending an event message.
RAISE_DBG_INT	1	A debug interrupt has been caused by a write to DSR1.
TRIGGER	2	The trigger_in pin has been asserted.
DBG_IBREAK	3	A debug instruction address breakpoint triggered.
DBG_SBREAK	4	An enabled debug software breakpoint triggered.
DBG_DBREAK	5	A debug data breakpoint on a data address triggered.
RESERVED	6-7	Reserved.

11 Memory translation and protection

The ST240 provides full memory translation and protection by means of a translation lookaside buffer (TLB).

The TLB enables a virtual-memory based OS such as Linux to use the ST240, while also supporting an OS that does not use virtual memory.

The ST240 memory management system permits multiple virtual address spaces. Each virtual address space has associated with it an address space identifier (ASID).

The ST240 memory management system enables memory pages to be marked with three different policies: cached, uncached and write combining uncached, as defined in *Table 45 on page 99*.

11.1 TLB overview

Table 42 provides details of the unified TLB (UTLB) and the two micro TLBs (DTLB and ITLB).

Table 42. TLB information

TLB	Size	Comment
UTLB	64 entries.	Fully associative unified TLB with hardware assisted replacement, managed by software. Stores translations for both the instruction and data caches.
DTLB	8 entries.	Fully associative micro data TLB with hardware least recently used (LRU) replacement, flushed by software. The translations are used for load, store, prefetch, purge, invalidation and flush of data addresses operations and purge of instruction addresses.
ITLB	4 entries.	Fully associative micro instruction TLB with hardware LRU replacement, flushed by software. The translations are used for instruction fetch.

The micro TLBs act as small caches that keep copies of the most recently used translations. Only translations that are shared or match the current ASID are loaded into the micro TLBs.

The micro TLBs perform address translations. If an address misses either micro TLB then the relevant micro TLB sends a request to the UTLB for the translation. If present in the UTLB, it is transferred into the relevant micro TLB. If the translation is not in the UTLB then an exception is raised.

When the UTLB is changed, the micro TLBs are not updated; they can be flushed under software control by means of the TLB_CONTROL register. See *Section 11.4.9: TLB_CONTROL register on page 103*.

The UTLB size can be determined either by reading the version register and the knowledge that the ST240 UTLB contains 64 entries, or by reading the LIMIT field of the TLB_REPLACE register after reset as shown in *Table 52 on page 101*.

11.2 Address space

This section describes physical and virtual addresses.

11.2.1 Physical addresses

The ST240 TLB supports a 32-bit (4 GB) physical address space. The format of the TLB entries allows future variants to support up to a 44-bit physical address space.

11.2.2 Virtual addresses

Virtual addresses are 32-bit. The TLB supports the use of four page sizes: 4 Kbyte, 8 Kbyte, 4 Mbyte and 256 Mbyte.

Virtual addresses above 0xFFFF0000 are mapped to control registers. TLB mappings of these virtual addresses are ignored.

11.3 Caches

The instruction and data caches are virtually indexed and physically tagged. In both cases the cache tag RAM lookup occurs in parallel with the TLB lookup.

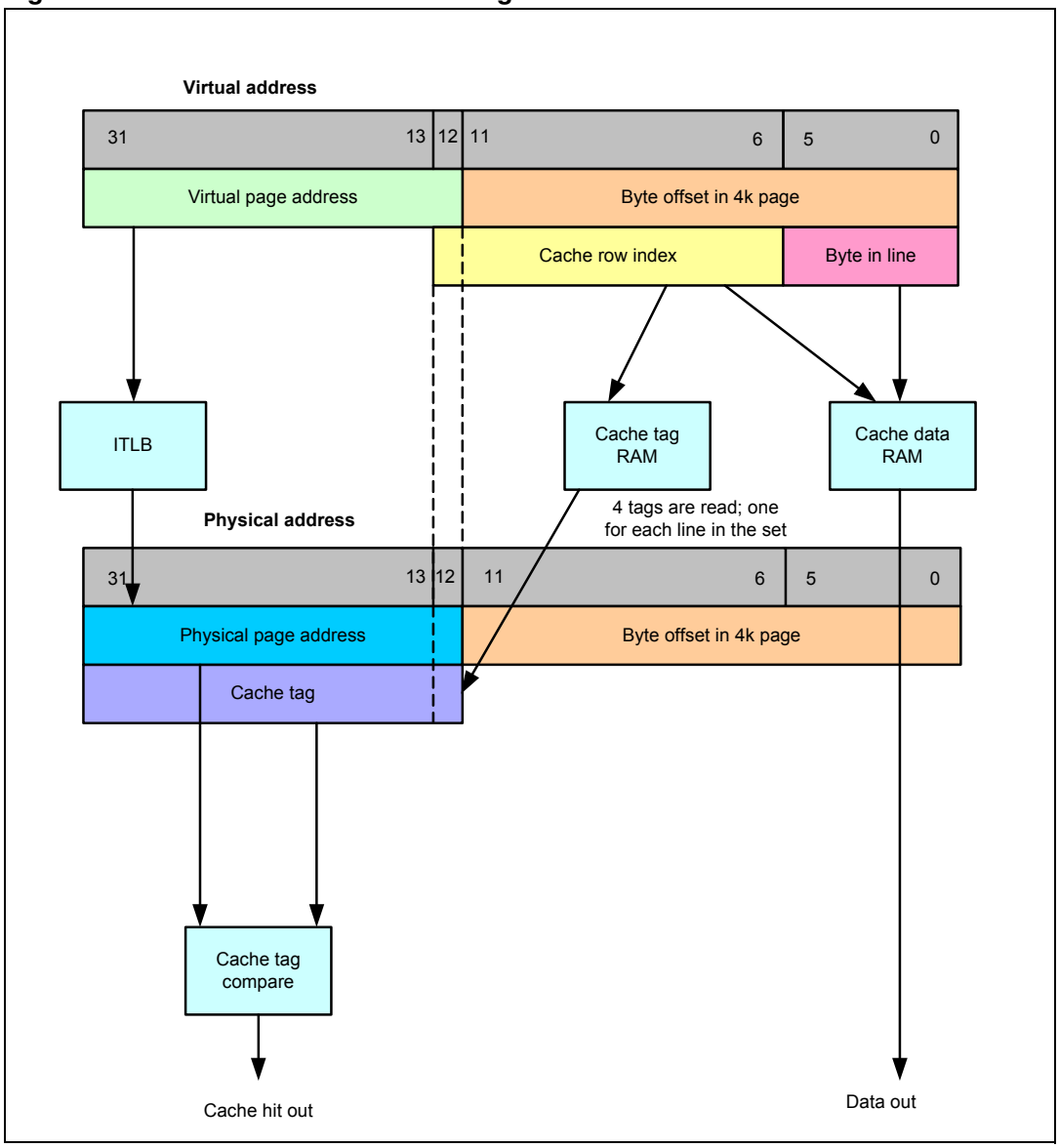
The arrangement of the instruction and data caches is described in the following sections. The only difference between the arrangement of the instruction and data caches is the linesize.

11.3.1 Instruction cache organization

Instruction cache addressing is illustrated in *Figure 22* where the cache access for a 4 Kbyte page is illustrated.

The instruction cache is 32 Kbyte 4-way set associative and built from 4 x 128 x 64 byte lines. The cache uses a round robin replacement policy with one replacement pointer per set of the cache.

Figure 22. Instruction cache addressing



In *Figure 22* the virtual address used to index the cache is split into three fields:

- bits that are translated: [31:12]
- cache line index bits: [12:6]
- byte offset within the cache line: [5:0]

The cache line index of the virtual address indexes the tag RAM to read the physical tags of all cache lines in the set. The ITLB translates bits [31:12] of the virtual address to produce the physical page address in *Figure 22*. Bits [31:12] of the physical page address are then compared against bits [31:12] of all four physical tags to check for a cache hit.

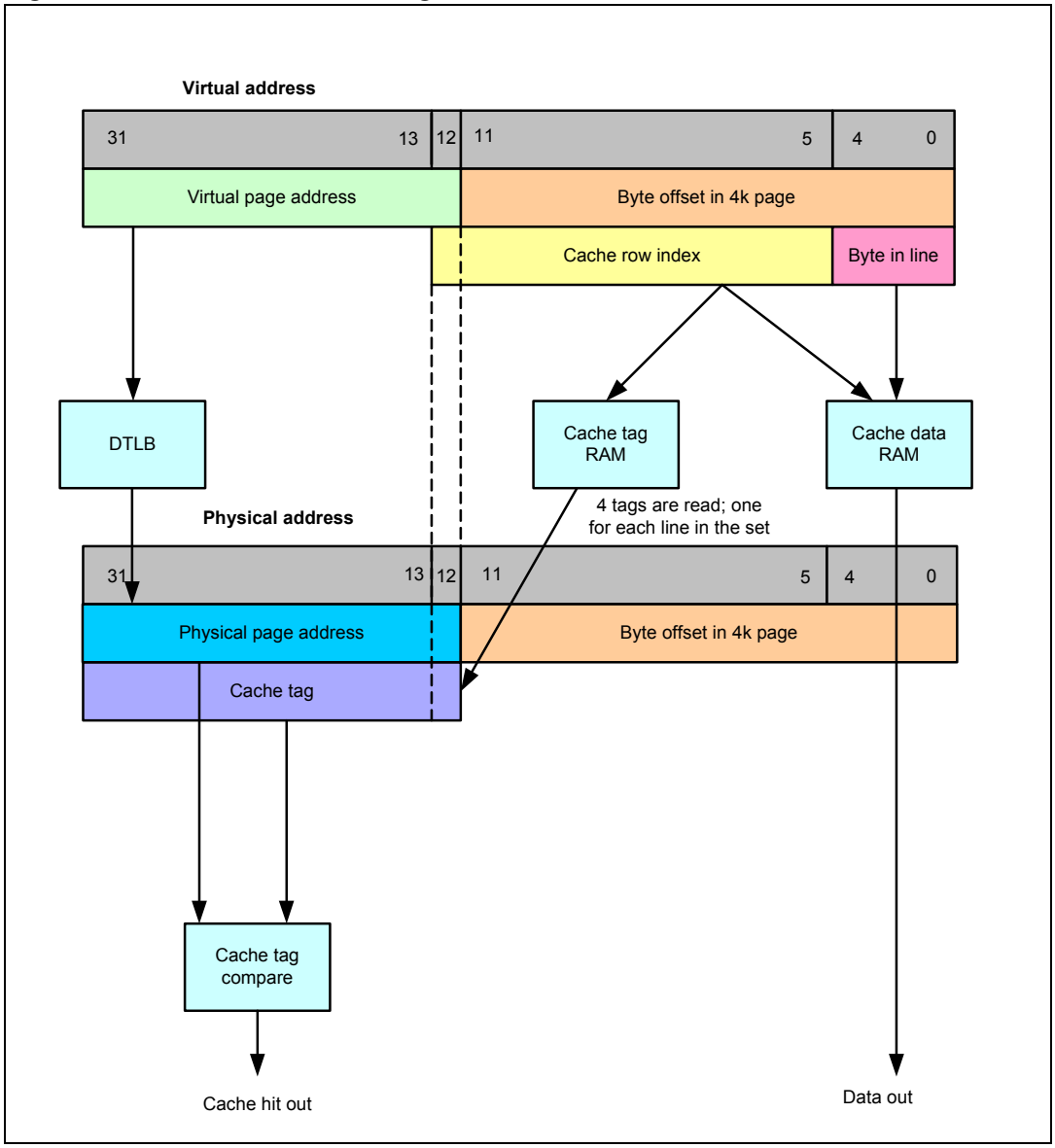
As bit 12 is translated and is also used for the cache line index, it may cause aliases to be generated, see *Section 11.3.3: Virtual aliases on page 97*.

11.3.2 Data cache organization

Data cache addressing is illustrated in *Figure 23* where the cache access for a 4 Kbyte page is illustrated.

The data cache is 32 Kbyte 4-way set associative and built from 4 x 256 x 32 byte lines. The cache uses a round robin replacement policy with one replacement pointer per set of the cache.

Figure 23. Data cache addressing



In *Figure 23* the virtual address used to index the cache is split into three fields:

- bits that are translated depending upon the minimum page size: [31:12]
- cache line index bits: [12:5]
- byte offset within the cache line: [4:0]

The cache line index of the virtual address indexes the tag RAM to read the physical tags of all cache lines in the set. The DTLB translates bits [31:12] of the virtual address to produce the physical page address in *Figure 22*. Bits [31:12] of the physical page address is then compared against bits [31:12] of all four physical tags to check for a cache hit.

As bit 12 is translated and is also used for the cache line index, it may cause aliases, see *Section 11.3.3*.

11.3.3 Virtual aliases

Virtual aliases occur in a virtually indexed, physically tagged cache when both the following are true:

- the page size is smaller than the way size
- two or more virtual addresses, which map on to the same physical address, differ in any bits used to index the cache (cache row index)

In this case there would be multiple locations caching a single physical address and coherency would not be guaranteed. A further problem can arise when it is necessary to remove a physical address from a cache. In the presence of a virtual alias, it may be necessary to remove more than one cache entry.

In the ST240 virtual aliasing can occur in either the instruction cache or the data cache, only when the 4 Kbyte page size is used.

The prevention or handling of virtual aliases is left to software. The data cache coherency control operations (*Section 12.5.4: Data cache control operations on page 122*) and the instruction cache coherency operations (*Section 12.4.3: Instruction cache control operations on page 116*) may be used to remove entries (and hence aliases or potential aliases) from the corresponding cache. Note that aliases only occur in the level 1 cache and so the cache coherency operations which operate only on the level 1 cache should be used for efficiency.

11.4 Control registers

A full list of control registers is provided in *Chapter 15: Control registers on page 145*.

11.4.1 PSW

The TLB can be enabled and disabled by a bit in the PSW (see *Chapter 4: Architectural state on page 27*).

While address translation is disabled (TLB_ENABLE = 0):

- virtual addresses are not translated and are used directly as physical addresses
- all data accesses are made uncached
- no TLB exceptions are raised (except for DEBUG_VIOLATION)
- addresses from 0xFFFF0000 to 0xFFFFFFFF always access control registers and cannot be used to access the data cache or the system memory

11.4.2 UTLB access

The 64 UTLB entries are 128 bits in size and are not directly memory mapped; only one TLB entry is accessible at any one time and is selected by programming the TLB_INDEX register. The selected entry is accessible using the TLB_ENTRYx (x = 0, 1, 2, 3 or 4) registers, which are described in *Section 11.4.4: TLB_ENTRY0 register on page 98* and in the sections following.

Selecting an out-of-range UTLB register causes writes to the TLB_ENTRY registers to be ignored and reads to return undefined results.

11.4.3 TLB_INDEX register

Table 43 shows the mapping for the TLB_INDEX register.

Table 43. TLB_INDEX bit fields

Name	Bit(s)	Writable	Reset	Comment
ENTRY	[7:0]	RW	0x0	Determines which of the 64 TLB entries is mapped to the TLB_ENTRYx registers. Writing a value to this register that is greater than the maximum UTLB entry available has no effect (the UTLB is not updated).
Reserved	[31:8]	RO	0x0	Reserved.

When the TLB_INDEX register is written, subsequent read/writes to the TLB_ENTRYx registers are to the indicated UTLB entry.

11.4.4 TLB_ENTRY0 register

The TLB_ENTRY0 register maps bits [31:0] of the TLB entry. *Table 44* lists the fields of the TLB_ENTRY0 register, the fields are described in subsequent tables.

Table 44. TLB_ENTRY0 bit fields

Name	Bit(s)	Writable	Reset	Comment
ASID	[7:0]	RW	0x0	The ASID that owns this page.
SHARED	8	RW	0x0	True if the page is shared by multiple ASIDs.
PROT_SUPER	[11:9]	RW	0x0	A three bit field that defines the protection of this region in supervisor mode. See *Table 48*.
PROT_USER	[14:12]	RW	0x0	A three bit field that defines the protection of this region in user mode. See *Table 48*.
DIRTY	15	RW	0x0	Page is dirty. If this bit is 0, stores to this page (if write permission is enabled) raise a TLB_WRITE_TO_CLEAN exception.

Table 44. TLB_ENTRY0 bit fields (Continued)

Name	Bit(s)	Writable	Reset	Comment
POLICY	[19:16]	RW	0x0	Cache policy for this page. See *Table 45*.
SIZE	[22:20]	RW	0x0	Size of this page (also used to disable the page). See *Table 46*.
PARTITION	[24:23]	RW	0x0	Data cache partition indicator. See *Table 47*.
MP_COHERENCY_ENABLE	25	RO	0x0	When 1, enables cache coherency within an MP cluster for the current TLB page, for future implementations which support this. When 0 MP cache coherency is disabled.
Reserved	[31:26]	RO	0x0	Reserved.

Writing zero to TLB_ENTRY0 disables the page.

Table 45 lists the possible values of the POLICY field.

Table 45. TLB_ENTRY0.POLICY values

Name	Value	Comment
UNCACHED	0	Uncached mode. Memory access to an uncached page accesses memory directly.
CACHED	1	Cached mode. In this mode: – a read that misses the cache causes the cache to be filled – a write that hits the cache is written into the cache – a write that misses the cache is sent to the write buffer
WCUNCACHED	2	Write combining uncached. In this mode: – writes are sent to the write buffer – reads access memory directly
Reserved	[15:3]	Reserved. Causes uncached if selected.

Table 46 lists of the possible values of the SIZE field.

Table 46. TLB_ENTRY0.SIZE values

Name	Value	Comment
DISABLED	0	Page is disabled.
8K	1	8 Kbyte page
4MB	2	4 Mbyte page
256MB	3	256 Mbyte page
4K	4	4 Kbyte page
Reserved	[7:5]	Reserved. An entry here causes the page to be disabled.

Table 47 lists of the possible values of the PARTITION field (see also *Section 12.5.1: L1 data cache partitioning on page 118*):

Table 47. TLB_ENTRY0.PARTITION values

Name	Value	Comment
REPLACE	0	Place in the way specified by the replacement pointer and increment the replacement pointer.
WAY1	1	Place in way 1 only.
WAY2	2	Place in way 2 only.
WAY3	3	Place in way 3 only.

Table 48 lists of the possible values of the PROT_USER and PROT_SUPER fields:

Table 48. PROT_USER and PROT_SUPER values

Name	Value	Comment
EXECUTE	1	Execute (instruction fetch and instruction cache purge) permission.
READ	2	Read (load, prefetch, data cache purge and invalidate) permission.
WRITE	4	Write (store) permission.

11.4.5 TLB_ENTRY1 register

The TLB_ENTRY1 register provides access to bits [63:32] of the TLB entry, the fields in this register are listed in *Table 49*.

Table 49. TLB_ENTRY1 bit fields

Name	Bit(s)	Writable	Reset	Comment
VADDR	[19:0]	RW	0x0	The upper 20 bits of the virtual address. – For 256 Mbyte pages only the upper 4 bits of this field are significant. – For 4 Mbyte pages only the upper 10 bits of this field are significant. – For 8 Kbyte pages only the upper 19 bits of this field are significant. – For 4 Kbyte pages all 20 bits are significant. All non-significant bits are ignored when performing the memory translation.
Reserved	[31:20]	RO	0x0	Reserved.

11.4.6 TLB_ENTRY2 register

The TLB_ENTRY2 register provides access to bits [95:64] of the TLB entry, the fields in this register are listed in *Table 50*.

Table 50. TLB_ENTRY2 bit fields

Name	Bit(s)	Writable	Reset	Comment
PADDR	[19:0]	RW	0x0	The upper 20 bits of the physical address. – For 256 Mbyte pages, only the upper 4 bits of this field are significant. – For 4 Mbyte pages, only the upper 10 bits of this field are significant. – For 8 Kbyte pages, only the upper 19 bits of this field are significant. – For 4 Kbyte pages, all 20 bits are significant. All non-significant bits are ignored when performing the memory translation.
Reserved	[31:20]	RO	0x0	Reserved.

11.4.7 TLB_ENTRY3 register

The TLB_ENTRY3 register maps bits [127:96] of the TLB entry, the fields in this register are listed in *Table 51*. This register is reserved for future expansion of TLB attributes.

Table 51. TLB_ENTRY3 bit fields

Name	Bit(s)	Writable	Reset	Comment
Reserved	[31:0]	RO	0x0	Reserved.

11.4.8 TLB_REPLACE register

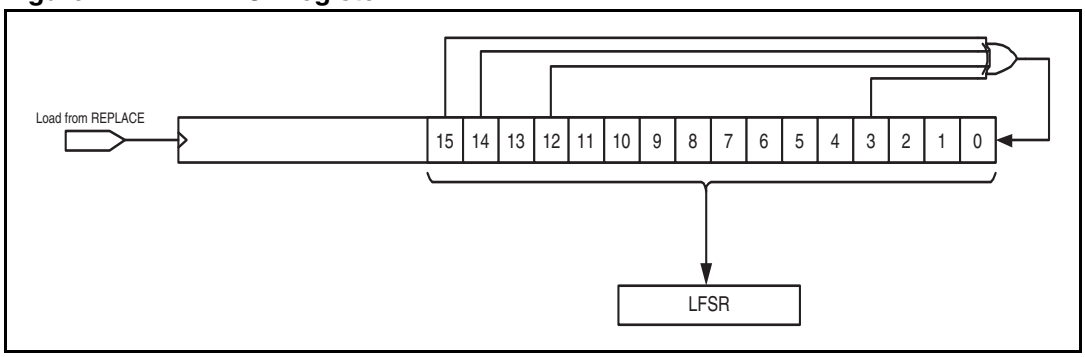
Table 52 shows the mapping of the TLB_REPLACE register.

Table 52. TLB_REPLACE bit fields

Name	Bit(s)	Writable	Reset	Comment
LFSR	[15:0]	RW	0xffff	Random number to determine the entry to replace next.
LIMIT	[23:16]	RW	0x40	Number of TLB entries to be replaced.
Reserved	[31:24]	RO	0x00	Reserved.

Figure 24 shows the structure of the REPLACE register.

Figure 24. REPLACE register



Software uses the replacement register to randomly select the TLB entry to replace. The value of the REPLACE field is generated in a pseudo-random manner using a 16-bit linear feedback shift register (LFSR) generating a maximum length sequence (taps on bits 3, 12, 14 and 15).

A read from the TLB_REPLACE register returns the current LFSR and LIMIT values. The LFSR is then clocked to generate a new value. The current value of the LFSR field can be changed by writing to the TLB_REPLACE register.

The LIMIT field is reset to the number of entries in the TLB. The LIMIT field can be changed by writing the TLB_REPLACE register. To reserve a number of entries for a fixed mapping, software sets the LIMIT field to less than the number of entries available to the TLB.

The LIMIT field is not used by the hardware, but is included to allow the software to determine quickly the next TLB entry to replace. A suggested replacement algorithm is:

1. Read the TLB_REPLACE control register into a general purpose register.
2. Extract the LFSR and LIMIT fields and perform an unsigned multiply of the two values.
3. Shift the result right 16 places.
4. Write the result to the TLB_INDEX register.

Step 2 can be achieved efficiently by using the **mullhu** operation.

11.4.9 TLB_CONTROL register

Table 53 shows the mapping of the TLB_CONTROL register.

Table 53. TLB_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
ITLB_FLUSH	0	RW	0x0	Writing a 1 to this bit flushes the entire ITLB. Writing a 0 to this bit has no effect. This bit always reads as 0.
DTLB_FLUSH	1	RW	0x0	Writing a 1 to this bit flushes the entire DTLB. Writing a 0 to this bit has no effect. This bit always reads as 0.
Reserved	[31:2]	RO	0x0	Reserved.

Before the ITLB or DTLB are flushed, the hardware ensures that all outstanding writes to the UTLB have completed.

11.4.10 TLB_ASID register

Table 54 shows the mapping of the TLB_ASID register.

Table 54. TLB_ASID bit fields

Name	Bit(s)	Writable	Reset	Comment
ASID	[7:0]	RW	0x0	Address space identifier. Writes to this register also cause the ITLB and DTLB to be flushed.
Reserved	[31:8]	RO	0x0	Reserved.

11.4.11 TLB_EXCAUSE register

When the ST240 raises a TLB exception, the TLB_EXCAUSE register is updated. Refer to *Figure 25: Instruction access on page 107* and *Figure 26: Data access on page 108* for the prioritization of the different TLB exception types.

The possible exceptions are listed in *Table 55*.

Table 55. TLB_EXCAUSE_CAUSE values

Name	Value	Comment
NO_MAPPING	0	Neither the relevant microTLB nor the UTLB had a mapping for the virtual address.
PROT_VIOLATION	1	An attempt has been made to violate the permissions of the page.
WRITE_TO_CLEAN	2	A write to a clean page has been attempted.
MULTI_MAPPING	3	There were multiple hits in the relevant microTLB or the UTLB. The software managing the TLB should ensure that this does not happen.

Table 55. TLB_EXCAUSE_CAUSE values (Continued)

Name	Value	Comment
DEBUG_VIOLATION	4	An illegal access to Debug RAM or DSRs addresses was attempted. Set purges do not cause this. This may be thrown when the TLB is disabled.
Reserved	5-7	Reserved.

The IN_UTLB field of TLB_EXCAUSE indicates whether the mapping which caused the exception was present in the UTLB at the time. Full details of the IN_UTLB field are shown in *Table 56*.

Access to debug resources which cause a DEBUG_VIOLATION are defined in *Section 20.1: Debug resource access on page 169*.

Table 56. TLB exception causes and the IN_UTLB bit

Cause	TLB is enabled?	Mapping is in UTLB?(1)	IN_UTLB
NO_MAPPING, PROT_VIOLATION, WRITE_TO_CLEAN	No	N/A	N/A
	Yes	Yes	1
		No	0
MULTI_MAPPING	No	N/A	N/A
	Yes		0
DEBUG_VIOLATION	No	N/A	0(2)
	Yes	Yes	1
		No	0

1. It is possible for a mapping to be present in one or more microTLBs but not in the UTLB, if the software removed a mapping from the UTLB and did not flush the relevant microTLB(s)
2. This is the only case where a TLB exception is thrown even though the TLB is disabled.

Table 57 describes the bit fields within the TLB_EXCAUSE register.

Table 57. TLB_EXCAUSE bit fields

Name	Bit(s)	Writable	Reset	Comment
INDEX	[7:0]	RW	0x0	TLB index of excepting page. Only valid when IN_UTLB field is 1.
Reserved	[15:8]	RO	0x0	Reserved.
CAUSE	[17:16]	RW	0x0	Cause of current TLB exception.
Reserved	18	RO	0x0	Reserved.
WRITE	19	RW	0x0	1 indicates that this exception was caused by an attempted store. 0 indicates that this exception was caused by an attempted load, purge or invalidation.

Table 57. TLB_EXCAUSE bit fields (Continued)

Name	Bit(s)	Writable	Reset	Comment
IN_UTLB	20	RW	0x0	1 the virtual address of the exception is mapped in the UTLB, and is not multimapped. When 1 the INDEX field is valid, when 0 it is not.
Reserved	[31:21]	RO	0x0	Reserved.

11.5 EXADDRESS register or TLB exceptions

When a DTLB exception is raised the EXADDRESS register contains the virtual address which caused the exception.

When an ITLB exception is raised the EXADDRESS register may contain either of the following virtual addresses, depending upon circumstances:

- the virtual address of the first syllable within the bundle that caused the exception if either of the following is true:
  - the bundle is fully contained within one TLB page
  - the bundle crosses a TLB page boundary and the syllable or syllables in the lower page cause a TLB exception
- the virtual address of a later syllable within the bundle if:
  - the bundle crosses a TLB page boundary and the syllable or syllables in the lower TLB page do not cause an exception although the first syllable in the higher TLB page does

11.6 TLB description

The TLB functionality is controlled completely by accessing the control registers provided.

11.6.1 Reset

After reset the contents of the UTLB are undefined. Before the TLB is enabled all entries must be programmed (or cleared) to prevent undefined behavior.

11.6.2 TLB coherency

The software must ensure coherency after making the following changes:

- changing the current ASID
- updating the UTLB

Changing the ASID

Changing the current ASID requires that all operations that may be affected by the change are flushed from the pipeline. This is achieved by executing a **syncins** or an **rfi.**

If it can be guaranteed that no operation within the next eight bundles following the bundle that changes the ASID are affected, then the **syncins**/**rfi** may be omitted.

Updating the UTLB

Updating the UTLB requires six cycles for the change to take effect. The change does not automatically update the micro TLBs.

If the properties of a virtual address are changed, operations in the pipeline using the old properties may be incoherent. In this case, the relevant micro TLB and the pipeline must be flushed to ensure coherency.

Flushing the pipeline is achieved by executing a **syncins** or an **rfi**. If updating the UTLB at the end of an exception handler, **rfi** is appropriate.

The recommended sequences for ensuring coherency are shown in *Table 58* and depend upon the pipeline length. An increased pipeline length in a future implementation will invalidate these sequences. **$r1** contains the value 1-3 depending on whether the ITLB or DTLB or both require flushing, see *Section 11.4.9: TLB_CONTROL register on page 103*.

Table 58. Ensuring coherency after UTLB updates

	Properties of VA changed	Properties of VA not changed

Normal	`stw TLB_CONTROL[$r0] = $r1;;`
	`nop;;`
	`syncins;;`
		`nop;;`
		`nop;;`
		`syncins;;`

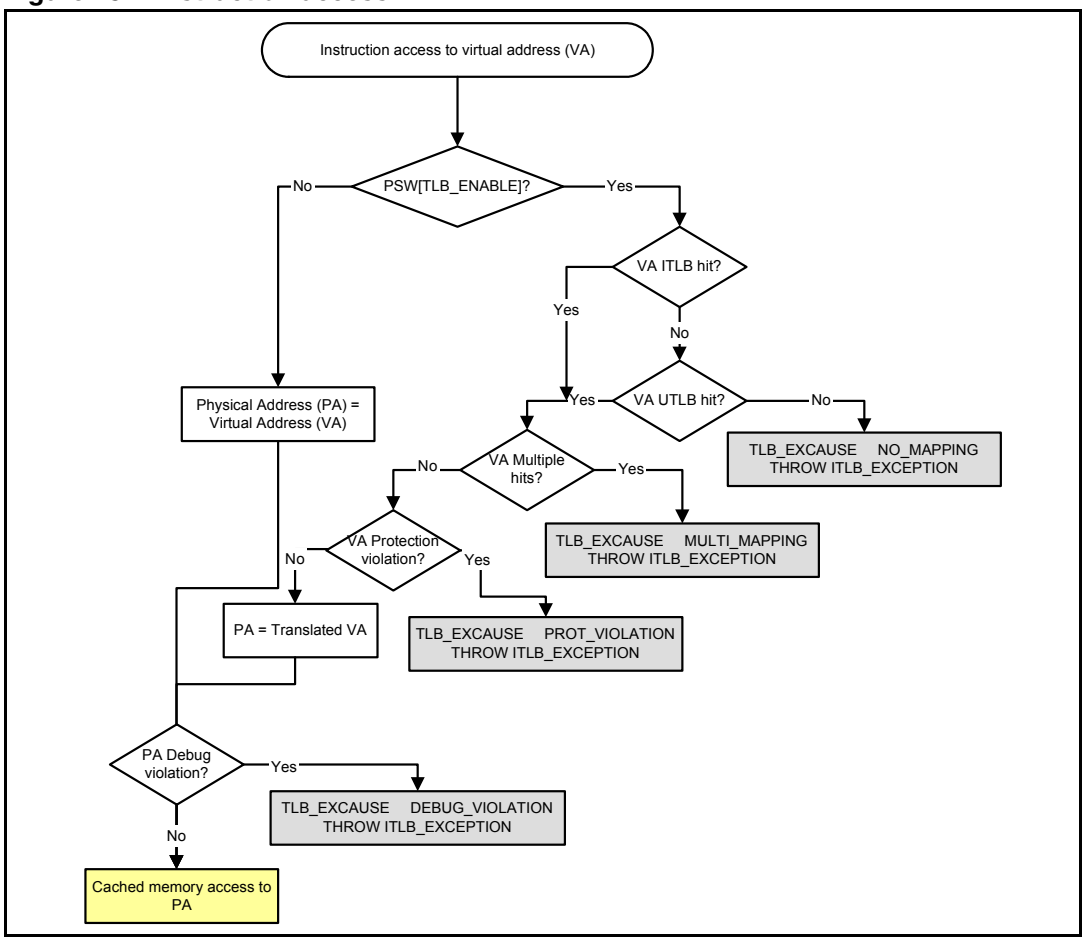
At the end of an exception handler	`stw TLB_CONTROL[$r0] = $r1;;` `rfi;;`	`nop;;` `rfi;;`

The normal sequence allowing the properties of the virtual address to change may be used to ensure coherency in any case.

11.6.3 Instruction accesses

Instruction accesses are always cached; the cache policy is ignored. The procedures for accessing instructions are summarized in *Figure 25*.

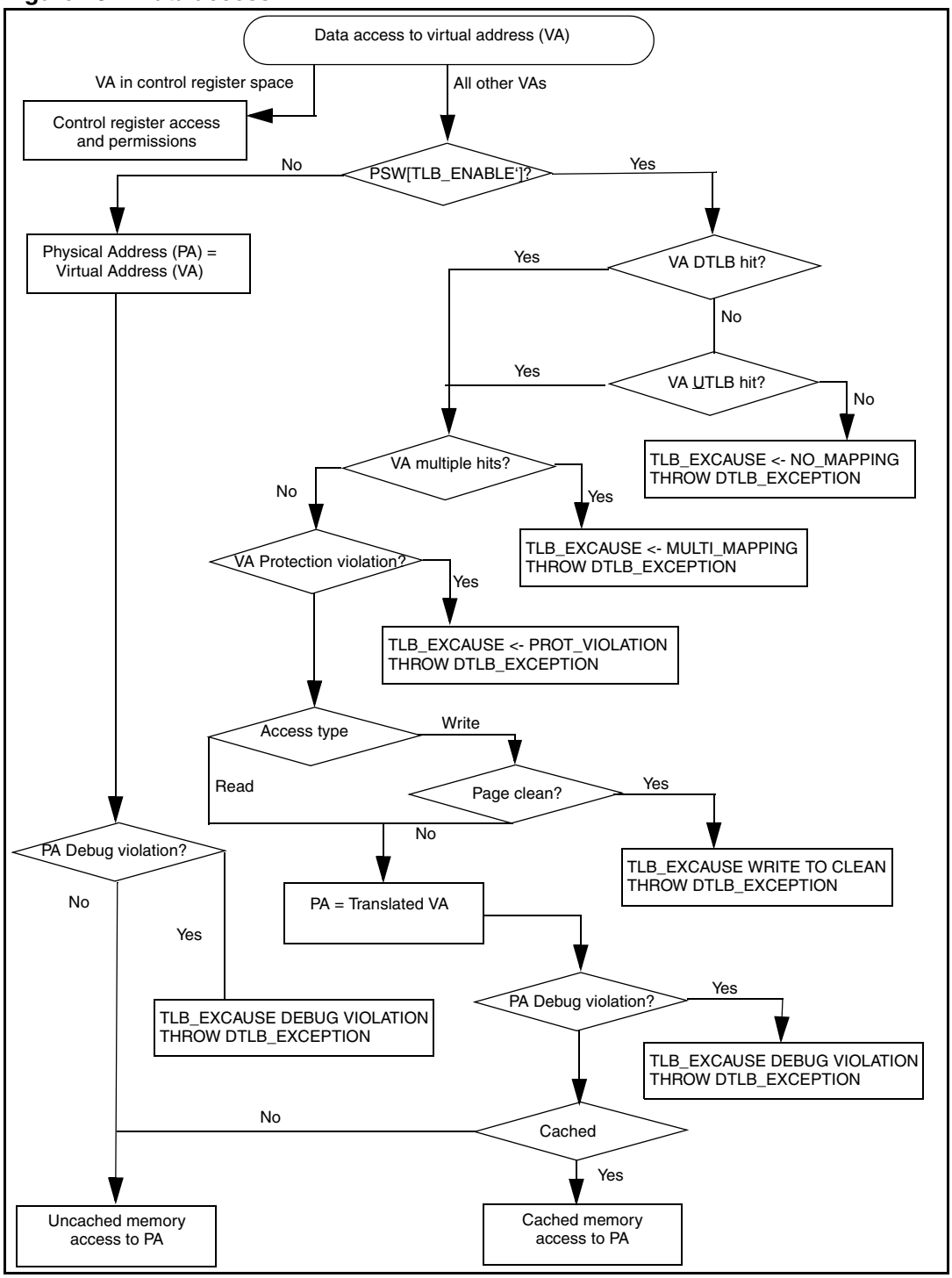
Figure 25. Instruction access



11.6.4 Data accesses

The procedures for reading and writing data are summarized in *Figure 26*.

Figure 26. Data access



Debug violations are described in *Section 20.1: Debug resource access on page 169*.

11.7 Speculative control unit

The speculative control unit (SCU) filters the physical addresses of prefetches that miss the cache to make sure that speculative STBus requests are not sent out to peripherals and unmapped memory regions.

The SCU supports four regions of memory aligned to the smallest TLB page size (4 Kbyte). If a prefetch operation is executed and the properties of the cache line addressed by the prefetch are such that an STBus request is required (according to *Section 12.5.2: Loads, stores and prefetches on page 119*), the SCU filters the physical address of the prefetch. If the physical address falls within one or more of the SCU regions, the cache line requested by the prefetch will be requested from the STBus; if not, the prefetch is silently aborted.

The regions are configured using the SCU_BASE*i* and SCU_LIMIT*i* control registers, where *i* = [0,3]. A region may be disabled by setting the base to be larger than the limit.

The SCU resets so each of the four regions cover the whole of memory.

11.7.1 SCU_BASE*i*, SCU_LIMIT*i* registers

The SCU_BASE*i* register defines the physical start address of the region where prefetches are permitted. This region is aligned to the smallest page size as only the 19 or 20 upper bits of the address can be programmed. The base address is inclusive, so setting BASE equal to LIMIT defines a 4 Kbyte region.

Table 59. SCU_BASE0 bit fields

Name	Bit(s)	Writable	Reset	Comment
BASE	[19:0]	RW	0x0	Represents the upper 20 bits of the base of this region.
Reserved	[31:20]	RO	0x0	Reserved.

Table 60. SCU_LIMIT0 bit fields

Name	Bit(s)	Writable	Reset	Comment
LIMIT	[19:0]	RW	0xfffff	Represents the upper 20 bits of the limit of this region.
Reserved	[31:20]	RO	0x0	Reserved.

11.7.2 Updates to SCU registers

To confirm that all STBus transactions before an SCU change are made with the old settings and all future transactions are made with new settings, a **sync** must be issued before updating the SCU registers.

12 Memory subsystem

This chapter describes the operation of the ST240 memory subsystem and the cache coherency control operations. The ST240 memory subsystem includes the following components:

- L1 instruction cache
- L1 data cache
- prefetch cache
- write buffer
- optional Data-side Tightly Coupled Memory (DTCM)
- optional external unified L2 cache

The presence or absence of the DTCM is documented in the system datasheet for a particular product, as is its size.

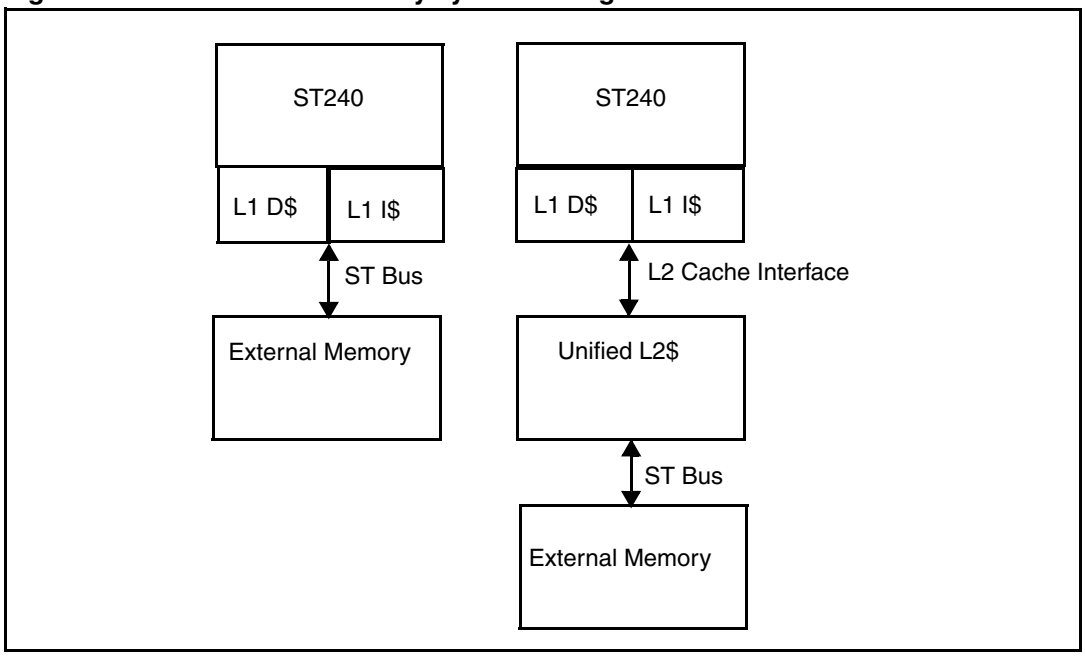
The memory subsystem is split broadly into two parts: the instruction side (I-side) and the data side (D-side). The I-side supports the fetching of instructions. The D-side supports the prefetching, storing and loading of data. Both sides support cache control operations to allow coherency.

Cache policies for loads and stores are determined by the Translation Lookaside Buffer (TLB), described in *Chapter 11: Memory translation and protection on page 93*. The use of TLB cache policy affects coherency as described in *Section 12.2: Memory coherency on page 113*.

12.1 Memory system configurations and terminology

This section discusses the two memory systems shown in *Figure 27*.

Figure 27. Documented memory system configurations



With reference to *Figure 27* the following terminology is used to explain the operation of the cache control operations on the memory hierarchy:

- memory: the external memory
- the lower memory system: any L2 cache and the external memory
- L1 caches: the L1 data cache and the L1 instruction cache

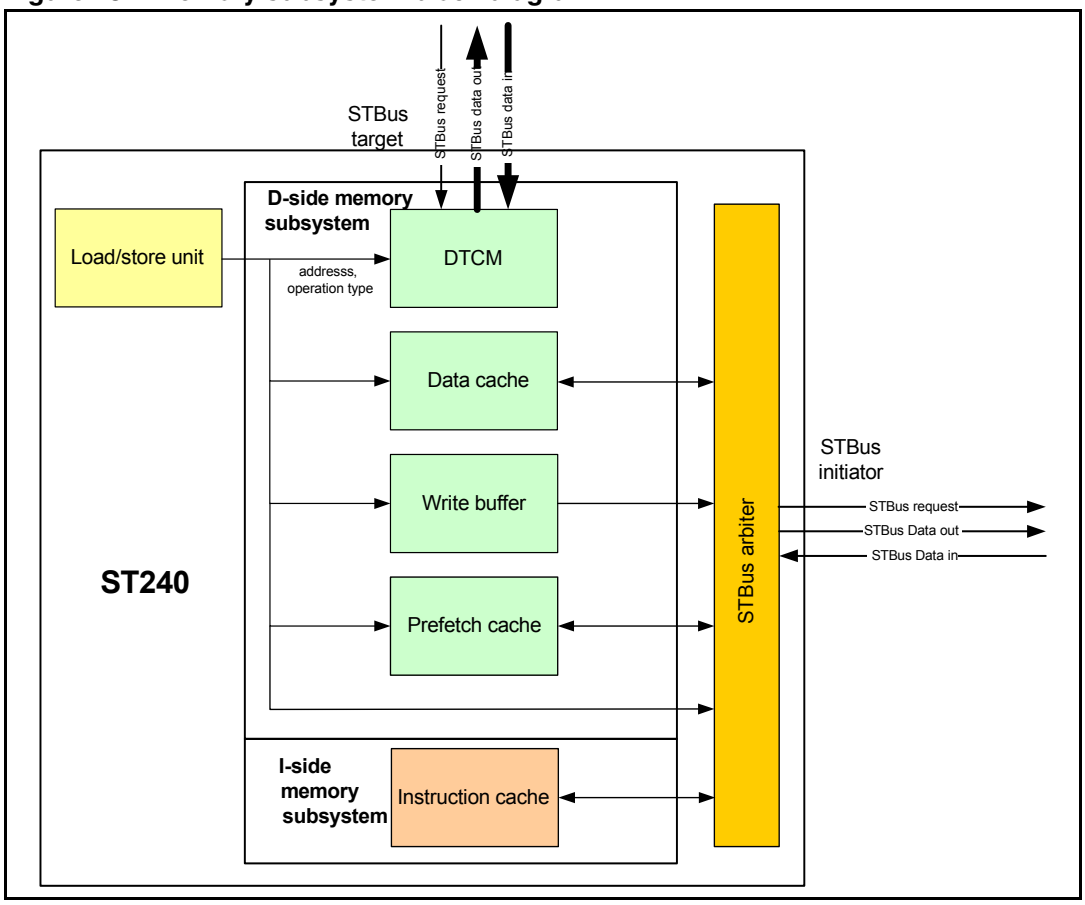
12.1.1 L2 cache coherency management

The ST240 supports a hardware interface for managing the coherency of an L2 cache. When the L2 cache is managed by the hardware, the ST240 will communicate purge, flush and invalidate requests to the L2 cache. The L2CACHE_DETAILS register (see *Table 65: L2CACHE_DETAILS bit fields on page 115*) indicates whether or not the L2 cache is managed by the hardware.

If the L2 cache is connected by a different interface, L2 cache coherency must be managed explicIty by software. Purge, flush and invalidate requests must be communicated to the L2 cache (typically through a register interface).

The ST240 memory subsystem is shown in simplified form in *Figure 28*.

Figure 28. Memory subsystem block diagram



With reference to *Figure 28*, which shows the memory subsystem hierarchy in simplified form, the following terminology is used:

- D-side memory subsystem: the L1 data cache, the write buffer, the prefetch cache and the optional DTCM
- I-side memory subsystem: the L1 instruction cache and any instruction fetch ahead logic

Cache control operations are available which:

- purge, flush or invalidate data from the L1 data cache, the prefetch cache and any L2 cache (**prgadd**, **flushadd**, **invadd**, **prgset**)
- purge, flush or invalidate only the L1 data cache and the prefetch cache (**prgadd.l1**, **flushadd.l1**, **invadd.l1**, **prgset.l1**)
- purge data from the L1 instruction cache and any L2 cache (**prginsadd**, **prginsset**)
- purge data only from the L1 instruction cache (**prginsadd.l1**, **prginsset.l1**)
- remove unexecuted syllables from the instruction fetch logic (**syncins**, **rfi**)

The following terminology is used:

- standard cache control operations: cache control operations which operate on the relevant L1 cache and any L2 cache
- L1 cache control operations: cache control operations which only operate on the relevant L1 cache

The cache control operations are described in *Section 12.4.3: Instruction cache control operations on page 116* and *Section 12.5.4: Data cache control operations on page 122*. The cache control operations are used for memory coherency as described in *Section 12.2: Memory coherency on page 113*.

12.2 Memory coherency

The ST240 ensures that data accesses mapped as cached in the TLB are coherent with other data accesses mapped as cached. The same is true of data access marked as either uncached or write-combining uncached. There is no guarantee of coherency:

- between the I-side and D-side memory subsystems
- between the I-side and D-side memory subsystems and the optional L2 cache
- between memory pages mapped with different cache policies
- between the ST240 and memory

To ensure coherency in the cases listed above data must be purged, flushed, invalidated and/or synchronized as covered in this section.

12.2.1 Instruction cache coherency

There are coherency issues associated with the I-side memory subsystem:

- incoherency between the L1 instruction cache, any L2 cache and the memory
- incoherency between the L1 instruction cache, the L1 data cache and any L2 cache
- incoherency between the L1 instruction cache, any L2 cache and the DTCM
- incoherency between the L1 instruction cache and fetched but unexecuted syllables

Incoherency between the L1 instruction cache, any L2 cache and memory

In this case the L1 instruction cache and (possibly) the L2 cache have incoherent copies of the data which is up to date in the memory.

In both cases the correct solution is to execute standard instruction cache purge operations. as described in *Purging the L1 instruction cache by address on page 116*. If it is known that the L2 copy is coherent with the memory, L1 instruction cache purges may be used instead for improved performance.

Incoherency between the L1 instruction cache, the L1 data cache and any L2 cache

In this case there is data incoherency between the L1 instruction cache, the L1 data cache and any L2 cache. The correct data is in the L1 data cache.

- The ST240 executes standard data cache purges or flushes as described in *Purging a memory range from the data cache on page 123* or *Flushing a memory range from the*

L1 and L2 data caches on page 124 to make the data visible to the L1 instruction cache.

- The ST240 executes standard instruction cache purges and removes any fetched but unexecuted syllables as described in *Purging a memory range from the data cache on page 123*.

If an L2 cache is in use, the performance may be optimized by using L1 data and instruction cache purge operations. Otherwise standard cache purge operations must be used.

Incoherency between the L1 instruction cache, any L2 cache and the DTCM

In this case there is data incoherency between the L1 instruction cache, any L2 cache and the DTCM. The correct data is in the DTCM.

The ST240 executes standard instruction cache purges and removes any fetched, but unexecuted syllables as described in *Purging a memory range from the L1 instruction cache and L2 cache on page 117*.

Incoherency between any fetched but unexecuted syllables and the L1 instruction cache

This case is a subset of the cases listed above. Instruction cache purges have been executed, but previously fetched syllables remain. The syllables are flushed out with a **syncins** or an **rfi** as described in *Purging a memory range from the L1 instruction cache and L2 cache on page 117*.

12.2.2 D-side coherency

There are coherency issues associated with the D-side memory subsystem:

- incoherency between the L1 data cache any L2 cache and the memory
- incoherency between the L1 data cache and the L1 instruction cache

Incoherency between the L1 data cache, any L2 cache and the memory

The ST240 has updated the L1 data cache and now the L2 cache and memory are incoherent. To update the L2 cache, the memory range is flushed or purged (using standard data cache operations) depending on whether the data is likely to be reused as described in *Flushing a memory range from the L1 and L2 data caches on page 124* and *Purging a memory range from the data cache on page 123*. If coherency with the memory is not required, L1 cache operations may be used.

A DMA engine has transferred data directly into the L2 cache. The ST240 makes the L1 data cache coherent by either purging or invalidating the incoherent memory range using L1 cache operations. If invalidations are used then any dirty data is discarded, so invalidation must be used with care.

A DMA engine has transferred data into the memory. The ST240 makes the L1 data cache and any L2 cache coherent by either purging or invalidating the incoherent memory range using standard cache operations. If invalidations are used, any dirty data is discarded, so invalidation must be used with care.

Incoherency between the L1 data cache and the L1 instruction cache

This case is covered in *Incoherency between the L1 instruction cache, the L1 data cache and any L2 cache on page 113*.

12.3 Cache information

Information about the caches is available in the read-only registers: ICACHE_LINESIZE, DCACHE_LINESIZE, ICACHE_SETS and DCACHE_SETS. The registers are described in *Table 61*, *Table 62*, *Table 63* and *Table 64*.

Table 61. ICACHE_LINESIZE bit fields

Name	Bit(s)	Writable	Reset	Comment
ICACHE_LINESIZE	[31:0]	RO	0x40	Number of bytes in one L1 instruction cache line

Table 62. DCACHE_LINESIZE bit fields

Name	Bit(s)	Writable	Reset	Comment
DCACHE_LINESIZE	[31:0]	RO	0x20	Number of bytes in one L1 data cache line

Table 63. ICACHE_SETS bit fields

Name	Bit(s)	Writable	Reset	Comment
ICACHE_SETS	[31:0]	RO	0x80(1)	Number of sets in the L1 instruction cache

1. Represents a 32 bytes instruction cache

Table 64. DCACHE_SETS bit fields

Name	Bit(s)	Writable	Reset	Comment
DCACHE_SETS	[31:0]	RO	0x100(1)	Number of sets in the L1 data cache

1. Represents a 32 Kbyte data cache

The L2CACHE_DETAILS register indicates whether the ST240 is managing an external L2 cache. The register bit fields are described in *Table 65*.

Table 65. L2CACHE_DETAILS bit fields

Name	Bit(s)	Writable	Reset	Comment
L2_HW_MANAGED	0	RO	System defined	When this bit is 1, it denotes that an L2 cache is present in the system and is managed by the ST240. When this bit is 0, an L2 may be present in the system, but it is the responsibility of the software to detect and manage it.
Reserved	[31:1]	RO	0x0	Reserved.

12.4 I-side memory subsystem

Instructions are always cached. The ST240 cannot execute uncached instructions.

12.4.1 L1 instruction cache

The instruction cache is described in *Section 11.3.1: Instruction cache organization on page 94*.

12.4.2 Instruction fetch

The ST240 fetches code from the L1 instruction cache and the implementation may choose to buffer unexecuted syllables.

The instruction cache only fetches syllables from one cache line at a time. The fetch of a bundle that crosses a cache line boundary therefore takes two cycles. The performance impact of the additional cycle is most visible when a branch target bundle crosses a cache line boundary, for example, at the start of a loop or a function.

If the TLB is enabled, instruction fetch requires execute permission.

If instruction fetch reaches the end of memory, the PC will silently wrap round.

12.4.3 Instruction cache control operations

Instruction cache coherency is discussed in *Section 12.2.1: Instruction cache coherency on page 113*.

Purging the L1 instruction cache by address

The **prginsadd** operation purges the specified virtual address from the L1instruction cache and the L2 cache, if there is one. **prginsadd.l1** only purges the L1 instruction cache. The address is byte aligned and one full cache line is purged. The size of the cache line is determined by the value in the ICACHE_LINESIZE register, see *Table 61 on page 115*.

If the virtual address passed to **prginsadd** or **prginsadd.l1** fails, the TLB makes the checks described in *Figure 22: Instruction cache addressing on page 95* and an ITLB exception is raised. There is a protection violation, if the page does not have execute permission; neither read nor write permission is required.

For implementational reasons **prginsadd** and **prginsadd.l1** use the DTLB to perform the address translation and TLB checks.

If **prginsadd** or **prginsadd.l1** is issued to a TLB page which is only used for instructions and not for data, an instruction related mapping becomes present in the DTLB. Similarly **prginsadd** and **prginsadd.l1** will cause DTLB related performance monitors to increment and not ITLB related performance monitors, see *Chapter 21: Performance monitoring on page 192*.

Purging a memory range from the L1 instruction cache and L2 cache

To purge a memory range from the L1 instruction cache and any L2 cache, carry out the following procedure:

1. Execute enough **prginsadd** operations to purge the entire memory range. The purge address is incremented by the line size each time; the line size is stored in the ICACHE_LINESIZE register, see *Table 61 on page 115*.
2. Flush the DTLB to remove any I-side mappings caused by the **prginsadd** operations, see *Section 11.4.9: TLB_CONTROL register on page 103*.
3. Execute a **syncins** or an **rfi** to ensure that any fetched, but unexecuted syllables have been removed.

Purging the L1 instruction cache and L2 cache by set

The **prginsset** operation uses bits from the address operand to obtain a cache row index, see *Section 11.3.1: Instruction cache organization on page 94*. The operation then purges all the cache lines within the set indicated by the cache row index. It also purges the required section of any L2 cache, so that when the L1 instruction cache has been purged so has the L2 cache. The replacement pointer of the set is reset to way zero. No translation or TLB checks are performed on the virtual address operand.

If an L2 cache is present and the L2 cache contains coherent data, **prginsset.l1** may be used, then the purges do not affect the L2 cache. In all other cases **prginsset** must be used.

To purge the instruction cache, carry out the following procedure:

1. Read the number of sets and the line size of the L1 instruction cache from the relevant control registers, see *Table 61 on page 115* and *Table 63 on page 115*.
2. Execute a **prginsset** for every set of the cache in turn. This is done by starting at zero, and incrementing the effective operand by the line size. The number of iterations is determined by the number of sets in the cache. The equivalent section of any L2 cache is also purged.
3. Execute a **syncins** or **rfi** to ensure that any unexecuted instructions have been flushed out. These operations also guarantee that the instruction cache purges have updated the L2 cache state as seen by the ST240.

If an L2 cache is present and the L2 cache contains coherent data, **prginsset.l1** may be used for performance reasons so that the purges do not affect the L2 cache. In all other cases **prginsset** must be used.

prginsset, **prgset.l1** and **syncins** can be executed in user mode but **rfi** cannot.

Note: It is inevitable that the sequence to purge the instruction cache will result in instruction cache misses. When the sequence completes the L1 instruction cache (and any L2 cache) will not be empty, part or all of the sequence will be present.

12.4.4 I-side STBus error

An L1 instruction cache refill request may cause an STBus error. An STBus error caused by a cache refill causes the cache line allocated to the refill to be invalidated and the replacement pointer not to be updated (so that the next fetch to the same set will allocate the same line again).

If a bundle from the cache line that caused the error reaches writeback, an STBUS_IC_ERROR exception is raised. Therefore I-side bus errors are synchronous events as they relate specifically to the PC which caused the error.

12.5 D-side memory subsystem

All data accesses take place through the D-side memory subsystem.

The data cache is described in *Section 11.3.2: Data cache organization on page 96*.

12.5.1 L1 data cache partitioning

L1 data cache partitioning enables the data cache either to operate as normal or to have one, two or three locked partitions, so the data cache can appear to be:

- a single 32 Kbyte four way cache (configuration 1)
- a 24 Kbyte three way cache and an 8 Kbyte direct mapped cache (configuration 2)
- a 16 Kbyte two way cache and two 8 Kbyte direct mapped caches (configuration 3)
- four 8 Kbyte direct mapped caches (configuration 4)

A combination of attributes in TLB entries and global state in the STATE1 control register controlling the data cache replacement pointers are used to achieve the cases listed above.

The global state restricts which ways of the cache are locked; that is, which way becomes a separate 8Kbyte direct mapped cache. Locked partitions can only be accessed by TLB pages that have been specifically programmed to access them.

Table 66 shows how to achieve the four cases listed above.

Table 66. Data cache partitioning control

Config	PARTITION in STATE1	TLB page 0	TLB page 1	TLB page 2	Other TLB pages	Comment
1	replace 0-3	replace	replace	replace	replace	default configuration
2	replace 0-2	way 3	replace	replace	replace	TLB page 0 references an 8 Kbyte locked way of the cache, other pages are restricted to ways 0 to 2.
3	replace 0-1	way 3	way 2	replace	replace	TLB pages 0, 1 each reference an 8 Kbyte locked way of the cache, other pages are restricted to ways 0 to 1.
4	replace 0	way 3	way 2	way1	replace	TLB pages 0, 1, 2 each reference an 8 Kbyte locked way of the cache, other pages are restricted to way 0.

The attributes in the TLB entries are controlled by the PARTITION field of TLB_ENTRY0 and described in *Table 47: TLB_ENTRY0.PARTITION values on page 100*.

Global control is achieved using the PARTITION field of STATE1, the entries in the PARTITION in STATE1 column are described in *Table 77: STATE1 bit fields on page 151*.

Changing the PARTITION field of the STATE1 register does not ensure that the replacement pointer lies in the specified range. Therefore if the global state is changed to restrict the replacement pointers, the entire L1 data cache must be purged as described in *Purging the L1 instruction cache and L2 cache by set on page 117*. This ensures that the replacement pointer for each set of the cache is set to zero. This purging is not required if the L1 data cache has not been used since the last reset.

The intended use of this facility is to lock part of the data cache that contains data that has no chance of eviction.

12.5.2 Loads, stores and prefetches

This section deals with loads, stores and prefetches.

Conditional loads, stores and prefetches

All load, store and prefetch operations can optionally refer to a condition bit. If the condition is true then the operation is executed. If the condition is not true the operation becomes a **nop** with no side-effects.

The conditional operations have a **c** suffix. For example, the **ldw** operation is not conditional, but the **ldwc** operation is.

Data widths

The following points apply when considering loads, stores and prefetches.

- All load and store operations work on data stored on the natural alignment of the data type; that is, long words on long word boundaries, words on word boundaries, half-word on half word boundaries.
- Load and store operations with misaligned addresses raise an exception that allows the implementation of misaligned loads by trap handlers.
- For a byte or half-word load, the data from memory is loaded into the least significant part of a register and is either sign-extended or zero extended according to the operation definition.
- For a byte or half-word store, the data is stored from the least significant part of a register.
- For a long word loads or stores, see *Long word accesses on page 119*.

Long word accesses

Long word data loads and stores access a contiguous register pair. Legal values for the first register in the pair are 0, 2, 4, 6...60.

Note that:

- pair 0 cannot be used as a destination register for **ldl** or **ldlc** as this is the encoding for prefetch
- if used as a source register for **stl** or **stlc**, the register pair is R0 and R0 (rather than R0 and R1) and so **stl** and **stlc** store 64-bits of 0
- register pair 62 cannot be used as a source for **stl** or **stlc** or destination for **ldl** or **ldlc** due to restrictions accessing the link register (R63)

The **ldl** and **ldlc** operations load a long word from memory:

- in little endian mode:
  - bits [31:0] are stored in the low register of the pair
  - bits [63:32] are stored in the high register
- in big endian mode:
  - bits [63:32] are stored in the low register of the pair
  - bits [31:0] are stored in the high register

The **stl** and **stlc** operations store a long word to memory:

- in little endian mode:
  - the low register of the pair is stored to bits [31:0]
  - the high register to bits [63:32]
- in big endian mode:
  - the low register of the pair is stored to bits [63:32]
  - the high register is stored to bits [31:0]

Cached loads and stores

Cached loads and stores are performed through the L1 data cache. Most cached memory operations operate on full cache lines:

- a load miss fetches a cache line from memory
- purges, flushes and invalidations operate on cache lines

The size of the cache line is determined as shown in *Table 62 on page 115*.

Stores that miss the cache do not operate on a full cache line, as cache lines are not allocated on write misses. Stores that miss the cache send the exact data width referenced by the store operation to the write buffer, see *Section 12.5.5: Write buffer on page 125*.

The memory subsystem presents a consistent view of cached memory to the ST240 programmer; that is, a store followed by a load to the same address always returns the stored data. To guarantee ordering of accesses to external memory in cached regions, purge or flush and synchronization operations must be used, see *Section 12.2: Memory coherency on page 113*.

Uncached load and stores

Uncached loads and stores are issued directly to the STBus. Data from an uncached region of memory is never brought into the L1 data cache or prefetch cache.

The precise amount of data specified in the access is transferred and the access is not combined with any other.

To guarantee that an uncached store has completed, either a **sync** operation (see *Section 12.5.3: Memory ordering on page 122*) or an uncached load to the same STBus target must be issued.[a]

a. This is true for a system using an STBus. A system using an AXI bus may not have this property, as loads and stores to the same address may be reordered. This is the rationale behind *Section 12.7: System bus requirements on page 131*

Write combining uncached loads and stores

Write combining uncached loads are equivalent to uncached loads. Write combining uncached stores are not issued directly to the STBus, but are sent to the write buffer instead, see *Section 12.5.5: Write buffer on page 125*.

Prefetching data

A **pft** or **pftc** operation is a hint to the memory subsystem that the given item of data may be accessed in the future. In the implementation, the prefetch cache stores the prefetched data and sends the data to the data cache when (and if) it is referenced by a load operation. Prefetched addresses are byte aligned, therefore it is not possible to execute a misaligned prefetch.

A prefetch is ignored (that is, treated as a **nop**) if either:

- it hits the data cache or the prefetch cache[b]
- it is issued when 8 other prefetches are outstanding, see *Prefetch performance notes on page 121*
- the address of the prefetch:
  - would cause a load operation to the same address to cause any TLB exceptions, except for DTLB MULTIMAPPING
  - maps to uncached or write combining uncached page in the TLB
  - maps to control register space
  - maps to the DTCM address space, if defined, see *Section 12.5.6: D-side tightly coupled memory on page 125*
  - does not fall into one of the valid regions in the SCU, see *Section 11.7: Speculative control unit on page 109*

A prefetch that does not fall into any of these categories is issued to the STBus. A prefetch which causes an STBus error will not raise an exception until a load is issued to the prefetched cache line.

Prefetches can cause DTLB MULTI_MAPPING and DBREAK exceptions.

Prefetch performance notes

The prefetch cache contains eight entries, each storing one cache line of data. The prefetch cache has an LRU (least recently used) replacement policy and will evict entries that contain valid data when more than eight requests are made. Prefetch requests are ignored when all eight entries are in the outstanding state (waiting for data). Therefore programs should avoid having more than seven prefetch operations between a prefetch and a load to the same address.

It is possible to raise a trap when prefetch evictions occur by using the PM_EVENT_PFTEVICTIONS performance counter, see *Chapter 21: Performance monitoring on page 192*.

b. This includes the case where the prefetch data is outstanding on the STBus.

12.5.3 Memory ordering

To enforce the completion and ordering of memory operations, use a **sync** or **dib** operation. One of these may be necessary after any purges, flushes or invalidations to ensure that the D-side memory subsystem is coherent with the L2 cache and/or the system memory.

The **sync** operation ensures that the lower memory system is up to date with respect to pending writes from the D-side memory subsystem. Pending writes are caused by store, purge or flush operations.

The **dib** operation ensures that any pending writes from the D-side memory subsystem are visible to the I-side memory subsystem. This includes data from cached, uncached and write combining uncached writes.

The uniprocessor ST240 **dib** performs a subset of the actions of the **sync** operation; **dib** sends all pending writes to the STBus and does not wait for responses. Therefore **dib** is a performance optimization over sync. For an MP ST240 the behavior of **dib** will be determined by the coherency system.

12.5.4 Data cache control operations

For discussion of data cache coherency see *Section 12.2.2: D-side coherency on page 114*.

Purge, invalidate and flush

The L1 purge (**prgadd.l1**) and L1 invalidate (**invadd.l1**) operations are used to ensure a copy of a particular data item is not cached in the D-side memory subsystem. An L1 purge operation writes dirty data back to the lower memory system and an invalidate operation does not. Therefore if an invalidate operation is used it must be certain that any dirty data can be safely discarded; that is, invalidation can lead to an incoherency.

The L1 flush (**flushadd.l1**) operation is used to ensure that cached data is consistent with the lower memory system. Dirty data is written to memory and the cache line is retained and becomes clean.

To ensure that purges, flushes and invalidations also operate on an L2 cache, if present, the **.l1** suffixes should be removed from the operations. Using the **.l1** operations may cause incoherences as they may not update the memory and so must be used with care.

The virtual address passed to purge, invalidation and flush operations are interpreted as byte addresses, so it is not possible for them to be misaligned. Any of these operations which hit the DTCM address space are executed as **nop**s.

Purging data by address

Purging data involves removing the specified virtual address from the relevant cache or caches. Any dirty data is written back to either the L2 cache or the memory.

The **prgadd** operation purges the specified virtual address from the L1 data cache and any L2 cache. The **prgadd.l1** operation only purges the L1 data cache. In both cases the address is byte aligned and one full cache line is purged.

The size of the cache line is determined as shown in *Table 62 on page 115*.

If the virtual address is in control register space or if the virtual address is in DTCM space, the purge operations have no effect. Otherwise if the virtual address fails the TLB checks described in*Figure 26: Data access on page 108*, a DTLB exception is raised and the state of the cache is not modified.

Note: There is a protection violation unless the page has read permission.

Invalidating data by address

Invalidating data involves removing the specified virtual address from the relevant cache or caches. Any dirty data is not written back to either the L2 cache or the memory.

The **invadd** operation invalidates the specified virtual address from the L1 data cache, the prefetch cache and any L2 cache. The **invadd.l1** operation only invalidates the L1 data cache and the prefetch cache. The address is byte aligned and one full cache line is invalidated.

If the virtual address is in control register space, or in DTCM address space the invalidate operations have no effect. Otherwise, if the virtual address passes the TLB checks described in *Figure 26: Data access on page 108*, a DTLB exception is raised and the state of the cache is not modified.

Note: There is a protection violation unless the page has read permission.

Flushing data by address

Flushing data involves removing the specified virtual address from the relevant cache or caches. If the virtual address hits a dirty line, the dirty data is written to the write buffer and the cache line becomes clean.

The **flushadd** operation flushes the specified virtual address from the L1 data cache and any L2 cache. The address is byte aligned and one full cache line is flushed. The **flushadd.l1** operation only flushes from the L1 data cache. The address is byte aligned and one full cache line is flushed.

If the virtual address is in control register space, or is in DTCM address space the flush operations have no effect. Otherwise, if the virtual address passes the TLB checks described in *Figure 26: Data access on page 108*, a DTLB exception is raised and the state of the cache is not modified.

Note: There is a protection violation unless the page has read permission.

Purging a memory range from the data cache

To purge a memory range from the L1 data cache and ensure that the state of the memory range in the L1 data cache is coherent with any L2 cache and memory, carry out the following sequence of operations:

1. Issue enough **prgadd**/**prgadd.l1** operations to ensure that the entire memory range is purged from the data cache. The purge address is incremented by the line size each time as shown in *Table 62 on page 115*.
2. Issue a **sync** to ensure that all purged dirty data has been written back to memory or issue a **dib** to ensure that the data is available for instruction fetch.

*Note: **dib** does not ensure that data is up to date in memory.*

If only the L2 cache needs to be updated, L1 purges can be used (**prgadd.l1**). In all other cases **prgadd** must be used.

Invalidating a memory range from the L1 and L2 data caches

To invalidate a memory range from the L1 data cache and any L2 cache, carry out the following operation:

1. Issue enough **invadd** operations to ensure that the entire memory range is invalidated. The invalidation address is incremented by the line size each time as shown in *Table 62 on page 115*.

If the purpose of the invalidation is to remove entries from the L1 data cache or prefetch cache (or both) without affecting the L2 cache, L1 cache purges can be used (**invadd.l1**). In all other cases **invadd** must be used.

Flushing a memory range from the L1 and L2 data caches

To flush a memory range from the L1 data cache and any L2 data cache, carry out the following sequence of operations:

1. Issue enough **flushadd**/**flushadd.l1** operations to ensure that the entire memory range is flushed from the data cache. The flush address is incremented by the line size each time as shown in *Table 62 on page 115*.
2. Issue a **sync** to ensure that all purged dirty data has been written back to memory, see *Table 67 on page 126*.

If only the L2 cache needs to be updated then L1 flushes can be used (**flushadd.l1**). In all other cases **flushadd** must be used.

Purging the L1 and L2 data caches by set

The **prgset** operation uses bits from the address operand to obtain a cache row index, see *Section 11.3.2: Data cache organization on page 96*. This operation purges all the cache lines within the set indicated by the cache row index and the relevant section of any L2 cache. The replacement pointer of the set is reset to way zero. No translation or TLB checks are performed on the virtual address operand. In addition to purging a set, **prgset** invalidates the entire prefetch cache.

If only the L1 data cache and prefetch cache are to be purged without affecting the L2 cache then **prgset.l1** can be used. In all other cases **prgset** must be used.

Carry out the following sequence of operations:

1. Read the number of sets and the line size of the data cache from the relevant control registers, see *Table 62 on page 115* and *Table 64 on page 115*.
2. Execute a **prgset** or **prgset.l1** to every set of the cache in turn. This is done by starting at zero and incrementing by the line size. The number of iterations is determined by the number of sets in the cache.
3. Execute a **sync** to ensure that the all purged dirty data has been written back to main memory or a **dib** to ensure that the data is available for instruction fetch.

12.5.5 Write buffer

Stores that miss the L1 data cache and dirty lines that are evicted from the cache are held in the write buffer pending write back to the STBus. Stores to TLB pages mapped as write-combining uncached are also held in the write buffer.

Stores to different cache policies that hit the same write buffer entry cause undefined behavior.

The write combining behavior of the write buffer allows stores that map to the same cache line to be merged into fewer STBus transactions.

Write buffer performance notes

The write buffer stores four entries, each holding one cache line. The replacement policy is LRU. To avoid evictions, applications that make many stores should ensure that as many as possible are made within three of the four available entries before storing to other memory locations.

12.5.6 D-side tightly coupled memory

The ST240 optionally provides a block of RAM (the DTCM), which the core can access, and which can also be accessed from the STBus using the target port as shown in *Figure 28 on page 112*. The physical address is fixed and is determined by input pins.

The DTCM forms part of the system memory map. Any device connected to the STBus, including the L1 instruction cache, is able to access the DTCM.

Load and store operations to cached TLB pages access the DTCM directly without going through the data cache or going through the STBus. The latency of accessing the DTCM is the same as the latency of accessing the data cache. The DTCM forms a fixed part of the address map, and is always enabled, so it is not possible for both the data cache and the DTCM to contain data at the same address.

DTCM and TLB behavior

The behavior for different TLB cache policies is shown in *Table 68: Memory subsystem behavior on page 128*. STBus requests are made for all accesses which hit the DTCM address space and which are not mapped as cached.

Power efficiency with a DTCM

The ST240 has a mechanism that allows software to access only the DTCM and not the data cache; this is done in order to save power. This is achieved with a bit in the PSW called DTCM_ONLY (as described in *Section 4.4: Program status word on page 28*). Any load or store which misses the DTCM address space when DTCM_ONLY is set raises a trap. The trap handler startup sequence automatically clears the DTCM_ONLY bit so that DTCM_ONLY mode is disabled during the handler startup. The intention is that the handler then returns causing the re-execution of the load or store.

DTCM access in idle mode

When the ST240 is in idle mode, it is still possible to access the DTCM without having to exit idle mode. See *Chapter 16: Low power modes on page 153*.

12.5.7 D-side STBus errors

An STBUS_DC_ERROR exception is raised if the D-side memory subsystem causes a STBus error.

In the case of a load this exception is synchronous and associated with the load operation that caused the error (even if the data was prefetched and held in the prefetch cache). An STBus error caused by a cache refill causes the cache line allocated to the refill to be invalidated and the replacement pointer updated. Therefore the next allocation within the same set will not allocate the line allocated to the fetch that caused the error.

Conversely, an STBus error caused by a store operation can not be associated with the store operation which caused it. Therefore STBus errors caused by store operations are asynchronous. An STBus error that arrives when interrupts are disabled due to a handler saving the current context causes the context to be overwritten. It is not possible to recover from this condition.

The priority of these two cases are shown in *Table 34: Trap types and priorities on page 84*.

12.5.8 Level 2 cache support

The ST240 instruction set allows a unified L2 cache to be used as an extension of the L1 caches.

The presence of an L2 cache is made visible to the software by a control register, see *Table 65: L2CACHE_DETAILS bit fields on page 115*.

If an L2 cache is supported by an ST240 implementation, the following cache control operations act upon both the L1 and L2 caches:

prgadd, **invadd**, **flushadd**, **prgset**, **sync, dib, prginsadd, prginsset**

This allows application code which uses these operations to execute correctly whether or not an L2 cache is present.

The cache control operations are listed in *Table 67*.

Table 67. Operations that are executed on the L1 or L1 and L2 caches

operation	Action	Example use
prgadd	Purge address from data cache and L2 cache.	To make modified cached data visible in external memory.
prgadd.l1	Purge address from data cache only.	To move dynamically generated code from the L1 data cache into L2 cache for L1 instruction cache visibility.
prgset	Purge set from data cache, invalidate prefetch cache. Purge corresponding part of L2 cache.	To purge corresponding part of the L2 cache so that the entire L2 is purged when the entire L1 data cache is purged.
prgset.l1	Purge set from data cache, invalidate prefetch cache.	To remove incoherent data from the L1 data cache without affecting the L2 cache.
invadd	Invalidate address from data cache and L2.	To invalidate a shared memory buffer before initiating a DMA transfer into the buffer, which overwrites the previous data.

Table 67. Operations that are executed on the L1 or L1 and L2 caches (Continued)

operation	Action	Example use
invadd.l1	Invalidate address from data cache only.	To remove virtual aliases from the D-side memory subsystem. The data is re-fetched from the L2 cache.
flushadd	Flush address from data cache and L2.	To write data back to main memory and retain a copy.
flushadd.l1	Flush address from data cache only.	To ensure dirty data is written back to the L2 cache, for example, for generated code (as for **prgadd.l1** above).
dib	Ensure that all pending D-side writes are visible for instruction fetch	To make generated code available for instruction fetch.
sync	Synchronize the data cache and L2 with the system memory.	To ensure that the system memory is up to date following data cache flushes or purges.
prginsadd	Purge address from instruction cache and L2 cache.	To purge incoherent data from the L1 instruction cache and L2.
prginsadd.l1	Purge address from L1 instruction cache only.	To re-fetch code from the L2 cache into the L1 instruction cache.
prginsset	Purge set from L1 instruction cache and corresponding part of L2 cache.	Purge corresponding part of the L2 cache so that the entire L2 is purged when the entire L1 instruction cache is purged.
prginsset.l1	Purge set from L1 instruction cache only.	Purge corresponding part of the L2 cache so that the entire L2 is purged when the entire L1 instruction cache is purged.

Prefetching in the presence of an L2 cache

As mentioned in *Prefetch performance notes on page 121*, prefetch cache evictions can be costly for performance. The presence of an L2 cache effectively removes the size limit of the prefetch cache as prefetched data is also written into the L2 cache. Any limit imposed on the number of prefetches executed for performance reasons is caused by the size and associativity of the L2 cache.

12.5.9 Summary of D-side memory subsystem behavior

Table 68 lists the operations supported by the D-side memory subsystem and their behavior under different cache policies. If the DTCM is not present, ignore the DTCM column. If an L2 cache is connected, it follows the behavior shown except where L1 cache control operations are used.

It is possible to hit the data cache and the write buffer with a single cached access. Execution of a **flushadd** or **flushadd.l1** operation causes data to be present in both locations, if the address operand hits the cache and the line is dirty. Subsequent access to the address which hit both cache and write buffer cause the data cache hit to take priority.

Write-combining uncached data in the write buffer is not defined to cause multiple hits between write buffer and data cache or prefetch cache on a cached access. In the case that a cached access hits a write-combining uncached entry in the write buffer the behavior is undefined.

Table 68. Memory subsystem behavior

Operation	Policy	DTCM	L1 Cache	Write buffer	Prefetch	Result
Loads: **ldb, ldbu, ldh, ldhu, ldw, ldl, ldbc, ldbuc, ldhc, ldhuc, ldwc, ldlc**	Uncached	Miss	Miss	Miss	Miss	Load uncached from STBus.
					Hit	Behavior is architecturally undefined
				Hit		
			Hit clean			
			Hit dirty			
		Hit				Load uncached from STBus.
Loads: **ldb, ldbu, ldh, ldhu, ldw, ldl, ldbc, ldbuc, ldhc, ldhuc, ldwc, ldlc**	WC uncached	Miss	Miss	Miss	Miss	Load uncached from STBus.
					Hit	Behaviour is architecturally undefined
				Hit		Flush write buffer entry. Load uncached via STBus.
			Hit clean			Behaviour is architecturally undefined
			Hit dirty			
		Hit				Load uncached from STBus.
Stores: **stb, sth, stw, stl, stbc, sthc, stwc, stlc**	Uncached	Miss	Miss	Miss	Miss	Store uncached to STBus.
					Hit	Behaviour is architecturally undefined
				Hit		
			Hit clean			
			Hit dirty			
		Hit				Store uncached to STBus.
Stores: **stb, sth, stw, stl, stbc, sthc, stwc, stlc**	WC Uncached	Miss	Miss	Miss	Miss	Store to write buffer.
					Hit	Behaviour is architecturally undefined.
				Hit		Store to write buffer. If hit entry contains cached data then behaviour is architecturally undefined
			Hit clean			Behaviour is architecturally undefined.
			Hit dirty			
		Hit				Store uncached to STBus.

Table 68. Memory subsystem behavior (Continued)

Operation	Policy	DTCM	L1 Cache	Write buffer	Prefetch	Result
Loads: **ldb, ldbu, ldh, ldhu, ldw, ldl, ldbc, ldbuc, ldhc, ldhuc, ldwc, ldlc**	Cached	Miss	Miss	Miss	Miss	Fill cache line from STBus, Load data from cache
					Hit	Transfer data to cache. Load data from cache
				Hit		If write buffer entry contains cached data then flush write buffer line, Fill cache line, Load data from cache. If write buffer entry contains wc-uncached data the behaviour is undefined.
			Hit clean			Load data from cache
			Hit dirty			
		Hit				Load data from DTCM
Stores: **stb, sth, stw, stl, stbc, sthc, stwc, stlc**	Cached	Miss	Miss	Miss	Miss	Store data to write buffer
					Hit	Discard prefetch cache entry, Store data to write buffer
				Hit		Store data to write buffer. If write buffer entry contains wc-uncached data the behaviour is architecturally undefined
			Hit clean			Store data to cache and make cache line dirty
			Hit dirty			Store data to cache
		Hit				Store data to DTCM
prgadd **prgadd.l1**	All	Miss	Miss	Miss	Miss	No effect
					Hit	Discard prefetch cache entry
				Hit		No effect
			Hit clean			Invalidate cache line
			Hit dirty			Purge cache line to write buffer. Invalidate cache line. If purged data hits a write buffer entry which contains wc-uncached data the behaviour is architecturally undefined.
		Hit				No effect

Table 68. Memory subsystem behavior (Continued)

Operation	Policy	DTCM	L1 Cache	Write buffer	Prefetch	Result
prgset	All					Purge cache lines in set (dirty data is sent to the write buffer). Invalidate cache lines. Discard all prefetch cache entries. Reset replacement pointer to way 0. If purged data hits any write buffer entries which contain wc-uncached data the behaviour is architecturally undefined.
invadd **invadd.l1**	All	Miss	Miss	Miss	Miss	No effect
					Hit	Discard prefetch cache entry
				Hit		No effect
			Hit clean			Invalidate cache line
			Hit dirty			
		Hit				No effect
flushadd or **flushadd.l1**	All	Miss	Miss	Miss	Miss	No effect
					Hit	
				Hit		
			Hit clean			Send data to write buffer. Make cache line clean. If flushed data hits a write buffer entry which contain wc-uncached data the behaviour is architecturally undefined.
			Hit dirty			
		Hit				No effect
dib	All					Ensure that all pending d-side writes are made visible to the instruction fetch mechanism.
sync						Flush entire write buffer to memory, wait for all STBus transactions to complete.

Table 68. Memory subsystem behavior (Continued)

Operation	Policy	DTCM	L1 Cache	Write buffer	Prefetch	Result
pft, **pftc**	Cached	Miss	Miss	Miss	Miss	Check the address in the SCU. if it misses the SCU: The prefetch is discarded If it hits the SCU: Issue a fetch to the STBus for the relevant cache line
					Hit	Prefetch is discarded
				Hit		
			Hit			
		Hit				
	Uncached, WC Uncached					

12.6 Reset state

After reset, all lines in the L1 instruction cache and L1 data cache are marked as invalid. The write buffer and prefetch cache entries are marked as empty. The contents of the DTCM are not altered on reset.

12.7 System bus requirements

The D-side memory subsystem can be used with system buses other than the STBus. Any system bus connected to the ST240 must not reorder reads and writes to the same STBus target.

13 Multi-processor and multi-threading support

The ST240 architecture supports cache-coherent multi-processing (MP) and multi-threading (MT). It achieves this by providing operations to allow atomic sequences (**ldwl**, **stwl** and **waitl**), a write memory barrier operation (**wmb**) and a data/instruction barrier (**dib**).

The architectural state in this chapter only covers the uniprocessor (UP) case. Further architectural state is required for either an MP or an MT implementation.

13.1 Atomic sequence

Atomic read, write and wait operations are available. They are used to enable efficient locking of a item of data in an MP cache coherent system, or when executing multiple hardware threads on a single MT core.

13.1.1 Atomic sequence control register

The following sections reference the LOCK_ADDRESS control register and a lock. The lock refers to the LOCKED bit of the LOCK_ADDRESS register.

Table 69. LOCK_ADDRESS register bit fields

Name	Bit(s)	Writable	Reset	Comment
SHADOW_LOCKED	0	RO	0x0	0: the shadow lock is not set 1: the lock is set
LOCKED	1	RO	0x0	Lock status. 0: the lock is not set 1: the lock is set
Reserved	[4:2]	RO	0x0	Reserved.

The lock address, stored in the ATOMIC_ADDRESS control register, is aligned to a data cache line (32-bytes) to be consistent with an MP coherency system. Unused words in the cache line assigned to lock addresses may be used for other purposes, but updating them clears the locks of other processors or hardware threads as described later in this chapter.

13.1.2 Atomic sequence

The load word linked (**ldwl**) and store word conditional linked (**stwl**) operations perform the atomic read, modify and write of a word of data. The wait for link (**waitl**) operation is used to make the atomic sequence more efficient.

The execution of an **ldwl** operation sets the lock. When an **stwl** is executed the lock is tested.

- If set, then the store succeeds, the cache line is updated, success is indicated to the software and the lock is cleared.
- If clear, then the store fails, the cache line is not updated and failure is indicated to the software.

The lock may have been cleared for any of the reasons listed in *Section 13.1.3: Lock clearing mechanisms on page 133*.

The code below can be used to implement a simple atomic increment sequence.

```
atomic_increment::
## load the value from r2 (the lock address) and set the lock
ldwl    $r1 = [$r2]
;;
## increment the value
add     $r1 = $r1, 1
;;
## attempt to store the new value. If address r2 has been modified
## by someone else, the store will fail
stwl    $b0, [$r2] = $r1
;;
## The store failed (someone else updated address r2) so try again
brf     $b0, atomic_increment
;;
```

This code fragment is included to demonstrate the basic operation and is suitable for a UP core which needs to detect whether a sequence has been interrupted. *Section 13.1.6: Atomic sequence code on page 134* contains some suggested code for real-world use in an MP or MT system.

In the case of a failure, the software then executes the read, modify, write sequence again until the write succeeds. This sequence can be costly if several MP cores are attempting to gain the same lock at once; both in terms of active power of tightly executing a short sequence of code attempting to gain the lock and in coherency traffic. In an MT system it is useful for efficiency reasons to have a scheme for descheduling a thread which is waiting for a lock as otherwise the thread will continue execution with no chance of gaining the lock. Therefore the wait for link (**waitl**) operation is provided. This causes execution to stall until the lock becomes available in an MP core system. In an MT core **waitl** causes the issuing thread to be descheduled. In a UP **waitl** is a nop.

Note that multiple cores or hardware threads can have multiple locks set at once. Only the first core or hardware thread to write to the lock address succeeds and invalidates all other locks to the same lock address. For this reason writes from **stwl** operations must be strictly ordered.

A UP has only one lock.

13.1.3 Lock clearing mechanisms

The following events clear a lock.

- Any remote MP core executing any store operation to the lock address, where the store is to a TLB page mapped both as cached and as participating in the coherency system, see *Section 13.1.6: Atomic sequence code on page 134*.
- Any hardware thread executing any store operation to the lock address clears the lock of all hardware threads with matching lock addresses except for the issuing hardware thread, see *Section 13.1.6: Atomic sequence code on page 134*.
- Any trap (exception or interrupt).
- Execution of an **rfi** or **stwl** operation by the local MP core, issuing hardware thread or a UP.

Only the last two points are relevant to the UP. See *Section 10.7: Saving and restoring execution state on page 87* for more information on these last two points.

13.1.4 Lock clearing on trap and rfi

The lock is cleared on an interrupt to ensure that when the interrupt handler starts, the lock is not set. This is to prevent the following cases in an MP or MT system.

- If an **ldwl**/**stwl** sequence is interrupted and the interrupt handler restores execution to another **ldwl**/**stwl** sequence and the lock has not been cleared since being set by the first sequence, then the second sequence may falsely see a granted lock as it may have been attempting to lock a different address.
- If a kernel routine incorrectly returns to an **ldwl**/**stwl** sequence without having cleared the lock, then the sequence may falsely see a granted lock as it may have been attempting to lock a different address.

Additionally in the UP case **ldwl**/**stwl** are used to detect whether the execution of a piece of code has been interrupted. Therefore clearing the lock on interrupt or **rfi** is required.

13.1.5 Shadow lock

The **ldwl**, **stwl**, **rfi** and other store operations modify the SHADOW_LOCK bit of LOCK_ADDRESS in the same way as the lock is modified as described in *Section 13.1.3: Lock clearing mechanisms on page 133*. Trap startup, however, does *not* clear the shadow lock. This is to enable a deadlock caused by all processors or threads waiting for the same lock to be detected. The shadow lock is only provided to improve debug.

13.1.6 Atomic sequence code

A UP, MP or MT core can use the code fragment later in this section to implement locking of shared resources. Two levels of locks are used: a hardware lock and a software lock.

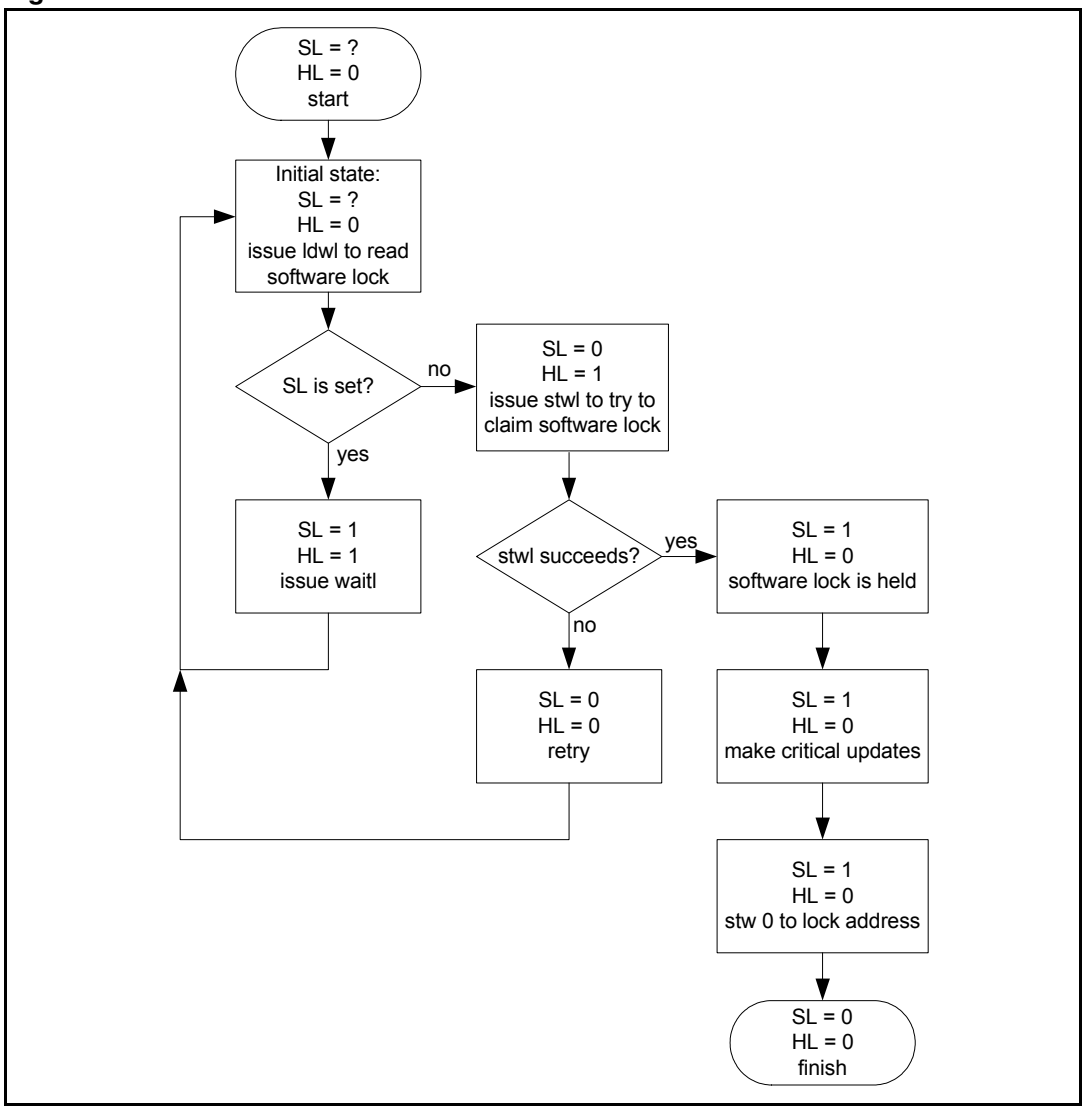
The software lock overrides the hardware lock. When the software lock is set, the hardware lock is irrelevant and must not be used. When the software lock is clear the hardware lock is used to gain the software lock.

The software lock is based upon the value stored at the lock address. A software convention is used in the example code to indicate whether the lock is set (`SPIN_LOCK_HELD_VALUE`) or if the lock is available (`SPIN_LOCK_AVAILABLE_VALUE`). The value of the software lock is loaded with an **ldwl**. If the software lock is set, no attempt is made to modify the data stored at the lock address (no **stwl** follows the **ldwl**) and a **waitl** is issued for efficiency. If the software lock is available, an **stwl** is used to attempt to claim the software lock.

If the software lock has already been taken, an **ldwl** must not be followed by a **stwl**.

Figure 29 is the software and hardware lock flowchart. In the diagram the software lock is represented by SL and the hardware lock by HL. A question mark (?) indicates that the value of the lock is not known at that point.

Figure 29. Software and hardware lock flow chart



Note that in *Figure 29* an **stw** is used to clear the software lock. This is because the hardware lock is clear at that point and so an **stwl** fails. The **stw** is required to clear the locks of all other hardware threads in the MT case and all other processors in the MP case, and therefore restarts execution in other threads/processors that have executed **waitl** operations. It is not required to clear the local hardware lock which is already clear at this point. Therefore this architecture chooses *not* to clear the hardware lock of the local MT hardware thread or the local MP core following a non **stwl** store to the lock address to simplify the implementation (any store operation excluding **stwl**).

In the code fragment LOCK_ADDRESS_RG is the register which contains the lock address.

```
__MP_spin_lock::
   ## load from the lock address and set the lock bit
   ldwl    $r1 = [ LOCK_ADDRESS_RG ]
   ;;
   ## if someone else has the software lock, indicated by $r1 =
   ## SPIN_LOCK_HELD_VALUE then don't attempt an stwl. The hardware
   ## lock may be set, and so the stwl may overwrite the software
   ## lock (although the value is the same). Also not executing an
   ## stwl reduces MP coherency traffic
   ##
   cmpeq   $b0 = $r1, SPIN_LOCK_HELD_VALUE
   mov     $r2 = SPIN_LOCK_HELD_VALUE
   ;;
   ## wait as someone else has the software lock
   br      $b0, __MP_spin_lock_wait
   ;;
   ## attempt to use the hardware lock to claim the software lock
   stwl    $b0, [ LOCK_ADDRESS_RG ] = $r2
   ;;
   ## did we get the hardware and software lock? If not retry.
   brf     $b0, __MP_spin_lock
   ;;
   ## LOCK_ADDRESS is set to SPIN_LOCK_HELD_VALUE
   ## so no-one else will try to update it
   ## other processors or hardware threads will execute waitls
   ## until the stw below clears their hardware locks
   <critical code section>
   ;;
   ## this is a normal store which will clear the hardware lock
   ## of other hardware threads/processes. An stwl would fail
   ## as the lock hardware lock was cleared by the stwl above
   stw [LOCK_ADDRESS_RG] = SPIN_LOCK_AVAILABLE_VALUE
   ;;
   ## function complete
   return $r63
   ;;
__MP_spin_lock_wait:
   ## we'll exit wait state when hardware lock is cleared by another
   ## core or hardware thread.
   waitl
   ;;
   ## wait state has exited as the hardware lock has been cleared
   ## restart lock acquisition sequence
   goto    __MP_spin_lock
   ;;
```

13.1.7 Atomic sequence semantics

ldwl

When a **ldwl** operation is executed:

- if the virtual address is in control register space, a CREG_ACCESS_VIOLATION is raised
- if no control register exceptions are raised, TLB checks are performed as defined in *Figure 26: Data access on page 108*
- if no exceptions are raised:
  - in the MP or MT case the physical address of the **ldwl** is stored to the upper bits of the LOCK_ADDRESS register (for a UP this does *not* occur)
  - the lock is set

stwl

When a **stwl** operation is executed:

- if the virtual address is in control register space, a CREG_ACCESS_VIOLATION is raised
- if no control register exceptions are raised, TLB checks are performed as defined in *Figure 26: Data access on page 108*
- if no exceptions are raised the lock is tested
- if the lock is set:
  - the **stwl** is executed as normal and the destination branch register is set to true
  - in the MP case the lock of all other processors that have a lock address that matches the address operand passed to **stwl** are cleared
  - in the MT case the lock of all hardware threads other than the issuing thread that have a lock address that matches the address operand passed to **stwl** are cleared
  - in the UP case there is no additional behavior
- if the lock is clear:
  - the **stwl** is not performed and the destination branch register is set to false
- the lock is always cleared

Note: ***stwl*** *never checks the lock address.*

Other store operations

When a store operation that is not an **stwl** is executed:

- if the virtual address is in control register space, a CREG_ACCESS_VIOLATION or CREG_NO_MAPPING may be raised, *Table 75: Control register spaces access exceptions on page 145*
- if no control register exceptions are raised then TLB checks are performed as defined in *Figure 26: Data access on page 108*
- in the MP case, if the store address matches the lock address of any other processor, the lock of that processor is cleared
- in the MT case, if the store address matches the lock address of any other hardware thread, the lock of that hardware thread is cleared
- in the UP case, the lock is not modified

waitl

When a **waitl** operation is executed:

- in the MP case, the lock is tested
  - if set, **waitl** stalls until either the lock is cleared or until an interrupt occurs
  - if clear, **waitl** is executed as a **nop**
- in the MT case, the lock is tested
  - if set, the issuing hardware thread deschedules and will not reschedule until either the lock is cleared or until an interrupt occurs
  - if clear, **waitl** is executed as a **nop**
- in the UP case, **waitl** is always executed as a **nop**

13.2 Write memory barrier

In the MP case **wmb** ensures that all pending memory writes have been broadcast to the coherency system and that the broadcasts have been acknowledged. The write buffer is also flushed to the STBus.

In the MT and UP cases **wmb** is a **nop** with no side effects.

13.3 Data/instruction barrier

In all cases **dib** ensures that all memory writes from all data-side memory subsystems are visible to all instruction-side memory subsystems.

In the UP and MT case **dib** causes all pending writes to be sent to the STBus and completes without waiting for the responses to be received. In the MP case the exact function of **dib** depends on the coherency system.

13.4 Operation summary

Table 70 summarizes the behavior of each operation in each use case.

Table 70. Summary of operation behavior for MP, MT and UP

	MP	MT	UP
ldwl	Load word and set lock.		
stwl (lock is set)	Store word. Clear lock of all other processor s that have a lock on the same lock address.	Store word. Clear lock of all other hardware threads that have a lock on the same lock address	Store word and clear lock.
stwl (lock is clear)	Do not store word.		
non stwl store	Store data and clear lock of all other processors, if store address matches lock address.	Store data and clear lock of all other hardware threads, if store address matches lock address.	Store data.
waitl (lock is set)	Prevent execution until lock is cleared or interrupted.		Executed as a **nop**(1).

Table 70. Summary of operation behavior for MP, MT and UP (Continued)

	MP	MT	UP
waitl (lock is clear)	Executed as a **nop**.		
wmb	Broadcast all outstanding memory writes and wait for responses.	**nop**	
dib	Broadcast all outstanding memory writes and make the writes visible to all I-side memory subsystems.	Send all pending writes to STBus. Ensure visibility of resulting data to I-side memory subsystem.	

1. A UP may either execute **waitl** as a **nop** or prevent execution until the lock is cleared or interrupted as for an MP or MT core. The first implementation of the ST240 executes **waitl** as a **nop**; future MT/MP implementations running in UP mode will prevent execution until the lock is cleared or interrupted.

13.5 Address translation

In the MP and MT cases, issuing **ldwl** operations with the TLB enabled causes the translated address to be written to the LOCK_ADDRESS register. This is a side effect of the atomic sequence support and can be used to perform virtual to physical address translation in a convenient way.

13.6 Control registers for MP support

The following control registers or control register fields are available for an MP core. MP implementations may support some or all of the features listed. The UP core has all referenced fields as zero and read only.

- The ID of the current core is read from the MP_CORE_ID register, see *Section 15.4: MP core ID register on page 151*.
- Cache coherency may be enabled and disabled on a page by page basis, see *Section 11.4.4: TLB_ENTRY0 register on page 98*.

14 Streaming data interfaces

The streaming data interfaces (SDI) provide a mechanism for attaching the ST240 to an external device that needs to transfer large amounts of data. Use of the SDI interfaces enables the ST240 to access the data without going through the data cache. Therefore SDI usage reduces data cache pollution, STBus traffic and does not require data cache misses to be serviced to load the data from the external device.

The ST240 implements four unidirectional SDIs: SDI 0 and 1 are inputs and SDI 2 and 3 are outputs. 32-bit data is transferred through these SDI “channels”.

The SDIs are accessed using control registers. In the naming of the control registers, *i* = [0,3]. Reads from the relevant SDI*i*_DATA register read from an input channel and stall if no data is available. Writes to the relevant SDI*i*_DATA register write to an output channel and stall if the channel is full. The stalls caused by these two cases may be either interrupted or timed out.

Data is communicated in order, in both directions. The n-th data item communicated arrives after the (n - 1)th item and before the (n + 1)th data item.

14.1 SDI control registers

See *Chapter 15: Control registers on page 145* for the address of the control registers. By default none of the registers may be accessed in user mode. The bit fields of the SDI*i*_CONTROL registers(*i* = 0 to 3) are shown in *Table 71* and *Table 72*. Note that *Table 71* applies to SDI0_CONTROL and SDI1_CONTROL and *Table 72* applies to SDI2_CONTROL and SDI3_CONTROL. All other registers contain a single 32-bit bit field.

Table 71. SDI0_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
PRIV	[1:0]	RW	0x0	Privilege bits.
RESETINPUT	2	RO	0x0	RESETINPUT acts as RESETREQUEST when slave, RESETACK when master.
RESETOUTPUT	3	RW	0x0	RESETOUTPUT, acts as RESETREQUEST when master, RESETACK when slave.
INPUTNOTOUTPUT	4	RO	0x1	INPUTNOTOUTPUT.
Reserved	5	RO	0x0	Reserved
MASTERNOTSLAVE	6	RO	System defined	MASTERNOTSLAVE.
TIMEOUTENABLE	7	RW	0x0	Time out enable and disable: 1: disable time out exceptions 0: enable time out exceptions.
Reserved	[31:8]	RO	0x0	Reserved

Table 72. SDI2_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
PRIV	[1:0]	RW	0x0	Privilege bits.
RESETINPUT	2	RO	0x0	RESETINPUT acts as RESETREQUEST when slave, RESETACK when master.
RESETOUTPUT	3	RW	0x0	RESETOUTPUT acts as RESETREQUEST when master, RESETACK when slave.
INPUTNOTOUTPUT	4	RO	0x0	INPUTNOTOUTPUT.
Reserved	5	RO	0x0	Reserved.
MASTERNOTSLAVE	6	RO	System defined	MASTERNOTSLAVE.
TIMEOUTENABLE	7	RW	0x0	Time out enable and disable. 1: disables time out exceptions 0: enables time out exceptions
Reserved	[31:8]	RO	0x0	Reserved.

The definition of the PRIV field is shown in *Table 73*.

Table 73. SDI*i*_CONTROL_PRIV values

Name	Value	Comment
PRIV_NOUSER	0	Access only allowed in supervisor mode.
PRIV_USER	1	Allow user access to data and ready register.
Reserved	2-3	Reserved (defaults to privilege no user).

Input and output channels have the same set of control registers. The definition of some registers vary depending on the direction of the channel as shown in *Table 74*. All registers in *Table 74* reset to zero.

The access column shows the access rights in user, supervisor and debug mode:

NA No access, any access cause a CREG_ACCESS_VIOLATION

RO Read only, writes silently ignored

RW Read/write

CF Configurable read/write or no access

CFRO Configurable read only (writes silently ignored) or no access

Where only one value is listed in the Access column it refers to all three modes. Where two values are listed the second value refers to supervisor and debug mode.

Table 74. SDI control registers

SDI register	Access	Definition
SDI*i*_CONTROL	NA/RW	Used to configure the SDI including enabling/disabling time outs as shown in *Table 71*.
SDI*i*_TIMEOUT	NA/RW	The number of cycles an SDI data access is allowed to stall before the channel times out raising an SDI_TIMEOUT exception. The channel becoming available for input or output causes this value to be reset to the value stored in SDI*i*_COUNT. Reads zero after the channel times out.
SDI*i*_COUNT	NA/RW	The value written into SDI*i*_TIMEOUT after successful SDI accesses.
SDI*i*_DATA	CF/RW	For an input channel: Input channel data register. A reads stalls if not data is available. Writing to register has no effect. For an output channel: Output channel data register. A write stalls if the channel is full. Reading from the register returns zero.
SDI*i*_READY	CFRO/RO	For an input channel: Whether a slot is available to be read from the input channel. Resets to 0. For an output channel: Whether a slot is available to be written to in the output channel. Resets to 1.

14.2 Exceptions, interrupts, reset and restart

This section describes interrupts, time outs and restarts (soft resets).

14.2.1 Interrupts

Stalls due to SDI accesses can be interrupted. In effect, the bundle attempting to access the SDI is trapped before being executed. If the bundle is re-executed before the channel has become available, the bundle stalls again. Availability of the channel can be checked in the handler by checking that the value of SDI*i*_TIMEOUT has been reset to the value stored in SDI*i*_COUNT, or by checking that SDI*i*_READY is non zero.

14.2.2 Time outs

Stalls due to SDI accesses may time out. In effect, the bundle attempting to access the SDI traps before being executed. If the bundle is re-executed before the channel has become available, the bundle is trapped again. To avoid this, the handler should first check channel availability by checking if the value of SDI*i*_TIMEOUT has been reset to the value stored in SDI*i*_COUNT, or by checking that SDI*i*_READY is non zero. If the channel is still unavailable, the handler should write a non zero value to the SDI*i*_TIMEOUT register before re-executing the bundle.

The time out exception is generated by the ST240 and not by the channel.

14.2.3 Restart (soft reset)

An SDI interface can have the ST240 as the master or slave. The only distinction is related to resetting the channel.

If the ST240 is a master, the reset sequence is as follows:

- the software writes a 1 to the RESETREQUEST bit in SDI*i*_CONTROL
- the reset is acknowledged by RESETACK being asserted (the software polls for this bit being set)
- the software clears RESETREQUEST
- the software polls for RESETACK being cleared

If the ST240 is a slave then the sequence is reversed. The software on the ST240 must check for RESETREQUEST being asserted, and then drives RESETACK accordingly.

After an SDI reset any data buffered by the core is discarded, consequently:

- an output channel becomes available
- an input channel becomes empty

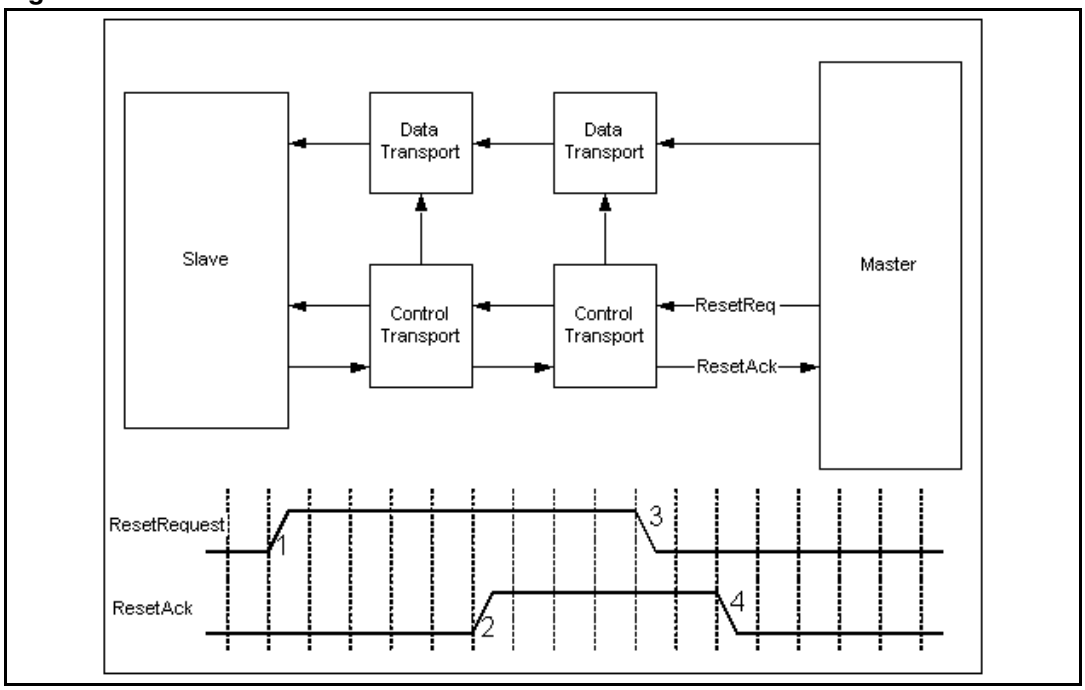
During the reset sequence the SDI*i*_READY and SDI*i*_DATA registers should not be accessed as the values read are implementation dependant. The reset sequence does not affect the contents of the SDI*i*_TIMEOUT and SDI*i*_COUNT registers.

The reset sequence is as follows:

1.	Master requests reset	Subsystem resets itself and consumes all data presented at inputs. RESETREQUEST is forwarded to other slave side subsystems.
2.	All units in reset	After subsystem has reset itself AND all slave side subsystems have sent RESETACK, RESETACK can be forwarded to master.
3.	Master requests leave reset	Unit forwards removal of RESETREQUEST to all slave-side subsystems. Unit leaves reset and stops consuming data.
4.	All units out of reset	On receipt of RESETACK from all subsystems, RESETACK is forwarded to master. System can restart.

The reset control structure is illustrated in *Figure 30*.

Figure 30. Soft reset control structure



15 Control registers

The ST240 control registers contain architectural state information that is not typically accessed by application code. This includes the TLB, PSW, exception registers and breakpoint registers.

The ST240 can only access control registers with word load and store operations. Control registers are not accessible from the STBus.

The control register space is defined by the range of virtual addresses from 0xFFFF 0000 to 0xFFFF FFFF.

Control register loads or stores are executed without reference to the TLB as shown in *Figure 26: Data access on page 108*.

15.1 Exceptions

Table 75 shows which operations may access control registers and the causes of the two related exceptions for illegal accesses.

Table 75. Control register spaces access exceptions

Operation	Legal cases	CREG_ACCESS_ VIOLATION	CREG_NO_MAPPING
ldw, **ldwc**, **stw**, **stwc**	Aligned address that maps to a control register	The operation does not have the correct access permissions, see *Section 15.2*.	Either: – the address is misaligned – aligned address that maps to a control register that does not exist
All other loads and stores	None	All	None

15.2 Control register addresses

Table 76 shows the addresses and access permissions of all control register addresses. The addresses are all relative to 0xFFFF 0000.

The access column shows the access rights in user, supervisor and debug mode:

NA No access, any access cause a CREG_ACCESS_VIOLATION

RO Read only, writes silently ignored

RW Read/write

CF Configurable read/write or no access

CFRO Configurable read only (writes silently ignored) or no access

Table 76. Control registers - BASE: CREG_BASE

Name	Offset	Access (U/S/D)	Reset	Comment
DBG_BT_DEST7	0x0b48	NA/NA/RO	Not reset	Branch trace buffer entry 7 destination PC
DBG_BT_SRC7	0x0b50	NA/NA/RO	Not reset	Branch trace buffer entry 7 source PC
DBG_BT_DEST6	0x0b58	NA/NA/RO	Not reset	Branch trace buffer entry 6 destination PC
DBG_BT_SRC6	0x0b60	NA/NA/RO	Not reset	Branch trace buffer entry 6 source PC
DBG_BT_DEST5	0x0b68	NA/NA/RO	Not reset	Branch trace buffer entry 5 destination PC
DBG_BT_SRC5	0x0b70	NA/NA/RO	Not reset	Branch trace buffer entry 5 source PC
DBG_BT_DEST4	0x0b78	NA/NA/RO	Not reset	Branch trace buffer entry 4 destination PC
DBG_BT_SRC4	0x0b80	NA/NA/RO	Not reset	Branch trace buffer entry 4 source PC
DBG_BT_DEST3	0x0b88	NA/NA/RO	Not reset	Branch trace buffer entry 3 destination PC
DBG_BT_SRC3	0x0b90	NA/NA/RO	Not reset	Branch trace buffer entry 3 source PC
DBG_BT_DEST2	0x0b98	NA/NA/RO	Not reset	Branch trace buffer entry 2 destination PC
DBG_BT_SRC2	0x0ba0	NA/NA/RO	Not reset	Branch trace buffer entry 2 source PC
DBG_BT_DEST1	0x0ba8	NA/NA/RO	Not reset	Branch trace buffer entry 1 destination PC
DBG_BT_SRC1	0x0bb0	NA/NA/RO	Not reset	Branch trace buffer entry 1 source PC
DBG_BT_DEST0	0x0bb8	NA/NA/RO	Not reset	Branch trace buffer entry 0 destination PC
DBG_BT_SRC0	0x0bc0	NA/NA/RO	Not reset	Branch trace buffer entry 0 source PC
DBG_BT_CONTROL	0x0bc8	NA/NA/RW	Refer to bit field in *Table 116*	Branch trace buffer Control Register
DBG_SBREAK_CONTROL	0xbff8	NA/NA/RW	0x0	Debug software breakpoint control
DBG_EXCAUSENO	0xc000	NA/NA/RO	0x0	Cause of the last debug trap represented as an integer
DBG_IBREAK_CONTROL	0xc008	NA/NA/RW	0x0	Debug instruction breakpoint control

Table 76. Control registers - BASE: CREG_BASE (Continued)

Name	Offset	Access (U/S/D)	Reset	Comment
DBG_IBREAK_UPPER	0xc010	NA/NA/RW	0x0	Debug instruction breakpoint upper address
DBG_IBREAK_LOWER	0xc018	NA/NA/RW	0x0	Debug instruction breakpoint lower address
DBG_DBREAK_CONTROL	0xc020	NA/NA/RW	0x0	Debug data breakpoint control
DBG_DBREAK_UPPER	0xc028	NA/NA/RW	0x0	Debug data breakpoint upper address
DBG_DBREAK_LOWER	0xc030	NA/NA/RW	0x0	Debug data breakpoint lower address
DBG_EXADDRESS	0xc038	NA/NA/RO	0x0	Virtual data address which caused a DBG_DBREAK trap
SCU_BASE0	0xd000	NA/RW/RW	0x0	Base address of prefetch region 0
SCU_LIMIT0	0xd008	NA/RW/RW	0xfffff	Limit address of prefetch region 0
SCU_BASE1	0xd010	NA/RW/RW	0x0	Base address of prefetch region 1
SCU_LIMIT1	0xd018	NA/RW/RW	0xfffff	Limit address of prefetch region 1
SCU_BASE2	0xd020	NA/RW/RW	0x0	Base address of prefetch region 2
SCU_LIMIT2	0xd028	NA/RW/RW	0xfffff	Limit address of prefetch region 2
SCU_BASE3	0xd030	NA/RW/RW	0x0	Base address of prefetch region 3
SCU_LIMIT3	0xd038	NA/RW/RW	0xfffff	Limit address of prefetch region 3
SDI0_DATA	0xe000	CF/RW/RW	0x0	SDI 0 data
SDI0_READY	0xe008	CFRO/RO/ RO	0x0	SDI 0 ready
SDI0_CONTROL	0xe010	NA/RW/RW	Refer to bit field in *Table 71*	SDI 0 control
SDI0_COUNT	0xe018	NA/RW/RW	0x0	SDI 0 count
SDI0_TIMEOUT	0xe020	NA/RW/RW	0x0	SDI 0 timeout
SDI1_DATA	0xe400	CF/RW/RW	0x0	SDI 1 data
SDI1_READY	0xe408	CFRO/RO/ RO	0x0	SDI 1 ready
SDI1_CONTROL	0xe410	NA/RW/RW	Refer to bit field in *Table 71*	SDI 1 control
SDI1_COUNT	0xe418	NA/RW/RW	0x0	SDI 1 count
SDI1_TIMEOUT	0xe420	NA/RW/RW	0x0	SDI 1 timeout
SDI2_DATA	0xe800	CF/RW/RW	0x0	SDI 2 data
SDI2_READY	0xe808	CFRO/RO/ RO	0x1	SDI 2 ready

Table 76. Control registers - BASE: CREG_BASE (Continued)

Name	Offset	Access (U/S/D)	Reset	Comment
SDI2_CONTROL	0xe810	NA/RW/RW	Refer to bit field in *Table 72*	SDI 2 control
SDI2_COUNT	0xe818	NA/RW/RW	0x0	SDI 2 count
SDI2_TIMEOUT	0xe820	NA/RW/RW	0x0	SDI 2 timeout
SDI3_DATA	0xec00	CF/RW/RW	0x0	SDI 3 data
SDI3_READY	0xec08	CFRO/RO/RO	0x1	SDI 3 ready
SDI3_CONTROL	0xec10	NA/RW/RW	Refer to bit field in *Table 72*	SDI 3 control
SDI3_COUNT	0xec18	NA/RW/RW	0x0	SDI 3 count
SDI3_TIMEOUT	0xec20	NA/RW/RW	0x0	SDI 3 timeout
PM_CR	0xf800	NA/RW/RW	0x0	Performance monitoring control
PM_CNT0	0xf808	NA/RW/RW	0x0	Performance monitor counter 0 value
PM_CNT1	0xf810	NA/RW/RW	0x0	Performance monitor counter 1 value
PM_CNT2	0xf818	NA/RW/RW	0x0	Performance monitor counter 2 value
PM_CNT3	0xf820	NA/RW/RW	0x0	Performance monitor counter 3 value
PM_PCLK	0xf828	RO/RW/RW	0x0	Performance monitor core cycle counter (lower 32 bits)
PM_PCLKH	0xf82c	RO/RW/RW	0x0	Performance monitor core cycle counter (upper 32 bits)
PM_INT	0xf830	NA/RW/RW	0x0	Performance monitoring interrupt control
IBREAK_CONTROL	0xfdc0	NA/RW/RW	0x0	Instruction breakpoint control
IBREAK_UPPER	0xfdc8	NA/RW/RW	0x0	Instruction breakpoint upper address
IBREAK_LOWER	0xfdd0	NA/RW/RW	0x0	Instruction breakpoint lower address
MP_CORE_ID	0xfdd8	NA/RO/RO	0x0	ID of the processor within an MP cluster
STATE1	0xfe00	NA/RW/RW	0x0	Global machine state register
TRAP_TLB	0xfe08	NA/RW/RW	0x0	PC that the core jumps to on a TLB exception
TRAP_INTERRUPT	0xfe10	NA/RW/RW	0x0	PC that the core jumps to on an interrupt

Table 76. Control registers - BASE: CREG_BASE (Continued)

Name	Offset	Access (U/S/D)	Reset	Comment
TRAP_BREAK	0xfe18	NA/RW/RW	0x0	PC that the core jumps to for breakpoint exceptions
TRAP_EXCEPTION	0xfe28	NA/RW/RW	0x0	PC that the core jumps to on all other exceptions
LOCK_ADDRESS	0xfe30	NA/RW/RW	0x0	Atomic load/store lock address
DCACHE_SETS	0xfe40	RO/RO/RO	0x100	Number of sets in the data cache
ICACHE_SETS	0xfe48	RO/RO/RO	0x80	Number of sets in the instruction cache
DCACHE_LINESIZE	0xfe50	RO/RO/RO	0x20	Data cache line size in bytes
ICACHE_LINESIZE	0xfe58	RO/RO/RO	0x40	Instruction cache line size in bytes
L2CACHE_DETAILS	0xfe60	RO/RO/RO	Refer to bit field in *Table 65*	Details of an L2 cache
DBREAK_CONTROL	0xfe70	NA/RW/RW	0x0	Data breakpoint control
DBREAK_UPPER	0xfe78	NA/RW/RW	0x0	Data breakpoint upper address
DBREAK_LOWER	0xfe80	NA/RW/RW	0x0	Data breakpoint lower address
TLB_ASID	0xff40	NA/RW/RW	0x0	Current address space identifier
TLB_REPLACE	0xff48	NA/RW/RW	0x40ffff	TLB replacement pointer
TLB_CONTROL	0xff50	NA/RW/RW	0x0	Control bits for TLB
TLB_EXCAUSE	0xff58	NA/RW/RW	0x0	Cause of the TLB related exception
TLB_ENTRY3	0xff60	NA/RW/RW	0x0	Bits [127:96] of the current TLB entry
TLB_ENTRY2	0xff68	NA/RW/RW	0x0	Bits [95:64] of the current TLB entry
TLB_ENTRY1	0xff70	NA/RW/RW	0x0	Bits [63:32] of the current TLB entry
TLB_ENTRY0	0xff78	NA/RW/RW	0x0	Bits [31:0] of the current TLB entry
TLB_INDEX	0xff80	NA/RW/RW	0x0	Index of the TLB entry pointed to by TLB_ENTRY0-3
EXCAUSENO	0xff88	NA/RW/RW	0x0	Cause of last normal trap represented as an integer
SCRATCH4	0xff90	NA/NA/RW	0x0	Scratch register reserved for use by the debug interrupt handler
SCRATCH3	0xff98	NA/RW/RW	0x0	Scratch register reserved for use by supervisor software
SCRATCH2	0xffa0	NA/RW/RW	0x0	Scratch register reserved for use by supervisor software

Table 76. Control registers - BASE: CREG_BASE (Continued)

Name	Offset	Access (U/S/D)	Reset	Comment
SCRATCH1	0xffa8	NA/RW/RW	0x0	Scratch register reserved for use by supervisor software
PERIPHERAL_BASE	0xffb0	NA/RO/RO	System defined	Base address of peripheral registers. The top 12 bits of this register are wired to the peripheral base input pins
SAVED_SAVED_PC	0xffb8	NA/RW/RW	0x0	PSW saved by as part of debug trap startup sequence
SAVED_SAVED_PSW	0xffc0	NA/RW/RW	0x0	PSW saved by as part of debug trap startup sequence
VERSION	0xffc8	NA/RO/RO	Refer to bit field in *Table 79*	Version number of the core
EXADDRESS	0xffd0	NA/RW/RW	0x0	Virtual data address in the case of either a DTLB, CREG or DBREAK trap. For other trap types this register is zero
EXCAUSE	0xffd8	NA/RO/RO	0x1	Cause of the last normal trap. Each normal trap type is indicated by a single bit in the register (a one-hot value)
SAVED_PC	0xffe8	NA/RW/RW	0x0	Saved PC, saved by trap handler startup sequence
SAVED_PSW	0xfff0	NA/RW/RW	0x0	Saved PSW, saved by trap handler startup sequence
PSW	0xfff8	NA/RO/RO	0x0	Program Status Word

15.3 Machine state register

The machine state register controls the global state of the machine.

Table 77. STATE1 bit fields

Name	Bit(s)	Writable	Reset	Comment
PARTITION	[1:0]	RW	0x0	Sets the maximum value for the round robin data cache replacement pointers as (3 - PARTITION). 00: Replace ways 0-3 01: Replace ways 0-2 10: Replace ways 0-1 11: Replace way 0 only A full data cache purge using **prgset** operations is required following the update of the field.
RESERVED.	[31:2]	RO	0x0	Reserved

For more details on cache partitioning see *Section 12.5.1: L1 data cache partitioning on page 118*.

15.4 MP core ID register

This register shows the hardware ID of a multi-processor core within a cache coherent cluster. For a uniprocessor core, it reads zero.

Table 78. MP_CORE_ID bit fields

Name	Bit(s)	Writable	Reset	Comment
MP_CORE_ID	[15:0]	RO	0x0	ID of an MP core within a cache coherent cluster for future implementations which support this.
Reserved	[31:16]	RO	0x0	Reserved

15.5 Version register

The VERSION register contains 3 fields which uniquely identify a particular release of the ST240. Refer to the datasheet for the meaning of each field.

Table 79. VERSION bit fields

Name	Bit(s)	Writable	Reset	Comment
PRODUCT_ID	[15:0]	RO	0x0000	Chip ID.
CORE_VERSION	[23:16]	RO	0x06	Core version number.
DSU_VERSION	[31:24]	RO	0x04	DSU design version number.

Note: The fields of the DSU version register (DSR0) are identical to the version register in *Table 79*.

16 Low power modes

The ST240 provides three low power modes:

- DTCM only mode
- idle mode
- retention mode

16.1 Low power operation with a DTCM

The ST240 provides a reduced power mode when running a program that has all of its data within the DTCM. See *Power efficiency with a DTCM on page 125*.

16.2 Idle mode

The ST240 enters idle mode by executing the **idle** macro (providing it is in a bundle by itself). When in idle mode, clocking is removed from the core (but not the peripherals) in order to save power.

The **idle** macro is encoded as a bundle containing a **goto** with zero immediate offset, that is: go to the same bundle and remain in the same bundle.

The **idle** macro is architecturally identical to the branch it derives from. When an interrupt or debug interrupt occurs, the core exits idle mode and jumps to the correct handler.

The idle macro can be inferred from C code by using

```
//enter idle mode
while (1);
```

16.2.1 Behavior in idle mode

When an **idle** macro is executed the ST240 completes the following operations in the order given:

1. Empties the pipeline, completing any operations issued before the **idle** macro.
2. Waits for all outstanding STBus transactions to complete.
3. Waits for the SDI output buffer to be become empty.
4. Enters idle mode.

*Note: The write buffer is not emptied, so all outstanding memory writes are not completed. If this is required then a **sync** must be executed before the **idle**.*

A bit is set in the PM_CR register to indicate to software that the core is currently in idle mode, see*Section 21.2: Control register (PM_CR) on page 195*, and this information is also available to the host by examining a DSU register.

The core aborts entry to idle mode and jumps to the correct handler on any of the following conditions:

- STBus error exception
- normal interrupt
- debug interrupt

The core exits idle mode and jumps to the correct handler on the following conditions:

- normal interrupt
- debug interrupt

While in idle mode:

- the only performance counter that increments is PM_EVENT_IDLE_CYCLES, see *Section 21.7: PM counters in idle mode on page 198*
- all peripherals continue to operate normally (timers, interrupt controller, DSU); that is, the STBus target port remains active
- access to the DTCM is still possible using the STBus target port, see *DTCM access in idle mode on page 125*
- the SDI input ports do not accept data
- the SDI output ports do not send out data (they must be empty before the core enters idle mode)

16.2.2 Latency of entry and exit of idle mode

The latency of entering idle mode depends upon the current state of the ST240, as shown in *Section 16.2.1: Behavior in idle mode on page 153*. If the ST240 is not waiting for an external interface to complete transactions (either SDI ports or STBus), entry takes only a few cycles. Exit always incurs a short delay.

16.3 Retention mode

The implementation of the ST240 may support retention mode.

Note: If the behavior of retention mode is architecturally visible, it will be defined in a later revision of this document.

17 Timers

The ST240 provides three timers. When enabled these timers continually count down to zero, reload and count down again. Interrupts are raised when the timers reach zero.

There are four registers associated with each timer; these are described in *Section 17.1: Timer registers on page 155*.

The timer interrupt lines are connected to internal interrupts 0, 1 and 2, see *Section 19.2: Interrupt registers on page 164*.

Note: As the timers are peripherals, the clock is likely to be slower than the clock to the core. Refer to the datasheet for the clock frequencies.

17.1 Timer registers

The timer registers are memory mapped; their addresses are listed in *Chapter 18: Peripheral addresses on page 157*.

Note: In the following sections, i = [0:2].

The timer register bit fields are described in *Table 80* to *Table 83*.

17.1.1 TIMECONST*i* register

The TIMECONST*i* register contains the value loaded into timer*i* after timer*i* reaches zero. The timer will reach zero after TIMECONST*i*+1 timer ticks. The bit fields of the TIMECONST*i* register are listed in *Table 80*.

Table 80. TIMECONST*i* bit fields

Name	Bit(s)	Writable	Reset	Comment
CONST	[31:0]	RW	0x0	Value to be reloaded when timer reaches zero.

17.1.2 TIMECOUNT*i* register

The TIMECOUNT*i* register returns the current value of the timer counter. The bit fields of the TIMECOUNT*i* register are listed in *Table 81*.

Table 81. TIMECOUNT*i* bit fields

Name	Bit(s)	Writable	Reset	Comment
COUNT	[31:0]	RW	0x0	Current value of timer counter

17.1.3 TIMECONTROL*i* register

The TIMECONTROL*i* register has three functions:

- enables the timer
- enables an interrupt when the timer reaches zero
- reads and resets the interrupt status of the timer

The bit fields of the TIMECONTROL*i* register are listed in *Table 82*.

Table 82. TIMECONTROL*i* bit fields

Name	Bit(s)	Writable	Reset	Comment
ENABLE	0	RW	0x0	Enable the timer.
INTENABLE	1	RW	0x0	Enable the timer interrupt.
STATUS	2	RW	0x0	Status of the timer interrupt. When 1, a timer has expired. Writing a 0 to this bit has no effect. Writing a 1 to this bit clears it.
Reserved	[31:3]	RO	0x0	Reserved

17.1.4 TIMEDIVIDE register

The TIMEDIVIDE register is associated with all three timers. The value in the DIVIDE field determines how many clock cycles are required for each timer tick and are therefore how many clock cycles are required to decrement the timer count. It is recommended that the TIMEDIVIDE register is set so that timer ticks occur every 1µs.

The bit fields of the TIMEDIVIDE*i* register are listed in *Table 83*.

Table 83. TIMEDIVIDE bit fields

Name	Bit(s)	Writable	Reset	Comment
DIVIDE	[15:0]	RW	0x0	Number of clock cycles required to decrement the timers + 1. A value of 0 causes the timers to decrement on every clock cycle.
Reserved	[31:16]	RO	0x0	Reserved.

18 Peripheral addresses

The interrupt controller, DSU and timers are all memory mapped peripherals. The DSU contains registers, the debug ROM and the debug RAM.

It is essential that the interrupt controller, DSU registers and timers are mapped in an uncached region in the TLB.

The core uses the debug ROM for instruction fetch and therefore it can be mapped in either a cached or uncached region.

The debug RAM can be mapped as cached or uncached memory as required.

18.1 Peripheral space address map

All addresses in *Table 84* are relative to the value in the PERIPHERAL_BASE register.

Table 84. Peripheral address map

Peripheral	Address range			Description
	Base	Allocated(1) size	Populated(2) size	
Interrupt controller and timer registers address space	0x0000	0x3000	0x1304	See *Table 87* for register addresses.
DSU register address space	0x3000	0x1000	0x0100	See *Table 88* for register addresses.
Debug ROM address space	0x4000	0x1000	0x0800	Default debug handler code in a 2 Kbyte ROM, see *Chapter 20: Debugging support on page 169*.
Debug RAM address space	0x6000	0x1000	0x0400	1 Kbyte of RAM only accessible in debug mode, see *Chapter 20: Debugging support on page 169*.

1. This column is used for debug resource protection checks
2. This column shows which addresses in the address space are valid. Not all bytes in the range are valid for address spaces containing registers.

Future implementations may increase the size of the debug ROM and debug RAM within the 4 Kbyte address space defined for each.

18.2 Peripheral access

The accesses allowed to peripherals are shown in *Table 85* and *Table 86*.

Accesses from the local core are shown in *Table 85*. The access types for the local core are:

- instruction fetch
- cached loads of different widths
- uncached loads of different widths

The address ranges are defined in *Table 84*.

Table 85. Peripheral access from local core

Peripheral	Permitted accesses	Non permitted access		
		Loads/instruction fetch which cause an STBus error	Stores which cause an STBus error	Stores which fail silently(1)
Timers and interrupt controller registers.	Uncached word loads and stores that map to a register.	All cached loads, all uncached loads except load word. Load words which do not map to a register. Instruction fetch.	All cached stores, all uncached stores except store word. Store words which do not map to a register.	None.
DSU registers.	Uncached word loads and stores that map to a register; subject to permission checks, see *Section 20.1 on page 169*.	All cached loads, all uncached loads except load word. Load words which do not map to a register. Instruction fetch.	All cached stores, all uncached stores except store word. Store words which do not map to a register.	None.
debug ROM.	Cached or uncached loads or instruction fetch within 2Kbyte of the base address.	Uncached load except load word. Any load above 2Kbyte from the base address.	Any stores except uncached word stores. Any store above 2Kbyte from the base address.	Uncached word stores within 2Kbyte of the base address.
debug RAM.	Full access in debug mode (see *Section 20.1 on page 169*) to addresses within 1Kbyte of the base address.	Any load above 1Kbyte from the base address.	Any store above 1Kbyte from the base address.	None.
Other address in peripheral space.	None.	All.	All.	None.

1. The resource is not updated and no STBus error is returned.

Accesses from STBus initiators other than the core are shown in *Table 86*.

Table 86. Peripheral access from other STBus initiators

Peripheral	Permitted accesses	Non permitted access		
		Loads which cause an STBus error	Stores which cause an STBus error	Stores which fail silently(1)
Timers and interrupt controller registers.	4-byte load/store that maps to a register.	Any load except load 4-bytes. Any 4-byte load that does not map to a register.	Any store other than store 4-bytes. Any 4-byte store that does not map to a register.	none.
DSU registers.	Load/store 4-bytes from a trusted source only and which maps to a register.	Any load except load 4-bytes. Any 4-byte load that does not map to a register.	Any store except store 4-bytes. Any 4-byte store that does not map to a register.	Store 4-bytes from non trusted STBus initiators which maps to a register.
debug ROM.	Load 4 or 32-bytes to address within 2Kbyte of the base address.	Loads except 4 or 32-bytes. Any load above 2Kbyte from the base address.	Any stores except store 4-bytes. Any store above 2Kbyte from the base address.	Store 4-bytes within 2Kbyte of the base address.
debug RAM.	No access.	All loads.	All stores.	None.
Other address in peripheral space.	None.	All.	All.	None.

1. The resource is not updated, and no STBus error is returned.

18.3 Peripheral addresses

The peripheral register addresses start from the peripheral base address, which can be found by reading the PERIPHERAL_BASE register, see *Chapter 15: Control registers on page 145*.

The access columns in *Table 87* and *Table 88* list the access rights for the listed registers. The acronyms used in this column are:

RO Read only, writes silently ignored.

RW Read/write.

CFRW Configurable read/write or read only.

CFRO Configurable read only (writes silently ignored) or no access.

18.3.1 Interrupt controller and timer registers

The interrupt controller and timer registers are listed in *Table 87*.

Table 87. Interrupt controller - BASE: INTCR_BASE

Name	Offset	Access	Reset	Comment
INT_INTPENDING0	0x0000	RO	0x0	Internal interrupt pending bits 31:0
EXT_INTPENDING0	0x0100	RO	0x0	External interrupt pending bits 31:0
EXT_INTPENDING1	0x0108	RO	0x0	External interrupt pending bits 63:32
INT_INTMASK0	0x0200	RW	0x0	Internal interrupt mask bits 31:0
EXT_INTMASK0	0x0300	RW	0x0	External interrupt mask bits 31:0
EXT_INTMASK1	0x0308	RW	0x0	External interrupt mask bits 63:32
INT_INTTEST0	0x0400	RW	0x0	Internal interrupt test register bits 31:0
EXT_INTTEST0	0x0500	RW	0x0	External interrupt test register bits 31:0
EXT_INTTEST1	0x0508	RW	0x0	External interrupt test register bits 63:32
INT_INTTESTSET0	0x0600	RW	0x0	Internal interrupt set test register bits 31:0
EXT_INTTESTSET0	0x0700	RW	0x0	External interrupt set test register bits 31:0
EXT_INTTESTSET1	0x0708	RW	0x0	External interrupt clear test register bits 63:32
INT_INTTESTCLR0	0x0800	RW	0x0	Internal interrupt clear test register bits 31:0
EXT_INTTESTCLR0	0x0900	RW	0x0	External interrupt clear test register bits 31:0
EXT_INTTESTCLR1	0x0908	RW	0x0	External interrupt clear test register bits 63:32
INT_INTMASKSET0	0x0a00	RW	0x0	Internal interrupt mask set bits 31:0
EXT_INTMASKSET0	0x0b00	RW	0x0	External interrupt mask set bits 31:0
EXT_INTMASKSET1	0x0b08	RW	0x0	External interrupt mask set bits 63:32
INT_INTMASKCLR0	0x0c00	RW	0x0	Internal interrupt mask clear bits 31:0
EXT_INTMASKCLR0	0x0d00	RW	0x0	External interrupt mask clear bits 31:0
EXT_INTMASKCLR1	0x0d08	RW	0x0	External interrupt mask clear bits 63:32
TIMECONST0	0x1000	RW	0x0	Timer constant
TIMECOUNT0	0x1008	RW	0x0	Timer counter
TIMECONTROL0	0x1010	RW	0x0	Timer control
TIMECONST1	0x1100	RW	0x0	Timer constant
TIMECOUNT1	0x1108	RW	0x0	Timer counter
TIMECONTROL1	0x1110	RW	0x0	Timer control
TIMECONST2	0x1200	RW	0x0	Timer constant

Table 87. Interrupt controller - BASE: INTCR_BASE (Continued)

Name	Offset	Access	Reset	Comment
TIMECOUNT2	0x1208	RW	0x0	Timer counter
TIMECONTROL2	0x1210	RW	0x0	Timer control
TIMEDIVIDE	0x1300	RW	0x0	Timer divide

18.3.2 DSU registers

The DSU registers are listed in *Table 88*. Most register have programmable write access configured by programming DSR1, see *Table 119: DSR1 bit fields on page 179*.

Table 88. Debug support unit - BASE: DSU_BASE

Name	Offset	Access	Reset	Comment
DSR0	0x000	RO	Refer to bit fields in *Table 79 on page 152*.	DSU version
DSR1	0x008	RW	Refer to bit fields in *Table 119 on page 179*.	DSU status
DSR2	0x010	RW	0x0	DSU output
DSR3	0x018	RW	0x0	DSU communication
DSR4	0x020	RW	0x0	DSU communication
DSR5	0x028	RW	0x0	DSU communication
DSR6	0x030	RW	0x0	DSU communication
DSR7	0x038	RW	0x0	DSU communication
DSR8	0x040	RW	0x0	DSU communication
DSR9	0x048	RW	0x0	DSU communication
DSR10	0x050	RW	0x0	DSU communication
DSR11	0x058	RW	0x0	DSU communication
DSR12	0x060	RW	0x0	DSU communication
DSR13	0x068	RW	0x0	DSU communication
DSR14	0x070	RW	0x0	DSU communication
DSR15	0x078	RW	0x0	DSU communication
DSR16	0x080	RW	0x0	DSU communication
DSR17	0x088	RW	0x0	DSU communication
DSR18	0x090	RW	0x0	DSU communication
DSR19	0x098	RW	0x0	DSU communication
DSR20	0x0a0	RW	0x0	DSU communication
DSR21	0x0a8	RW	0x0	DSU communication
DSR22	0x0b0	RW	0x0	DSU communication

Table 88. Debug support unit - BASE: DSU_BASE (Continued)

Name	Offset	Access	Reset	Comment
DSR23	0x0b8	RW	0x0	DSU communication
DSR24	0x0c0	RW	0x0	DSU communication
DSR25	0x0c8	RW	0x0	DSU communication
DSR26	0x0d0	RW	0x0	DSU communication
DSR27	0x0d8	RW	0x0	DSU communication
DSR28	0x0e0	RW	0x0	DSU communication
DSR29	0x0e8	RW	0x0	DSU communication
DSR30	0x0f0	RW	0x0	DSU communication
DSR31	0x0f8	RO	0x0	DSU virtual PC register

19 Interrupt controller

The ST240 interrupt controller supports up to 68 interrupt sources:

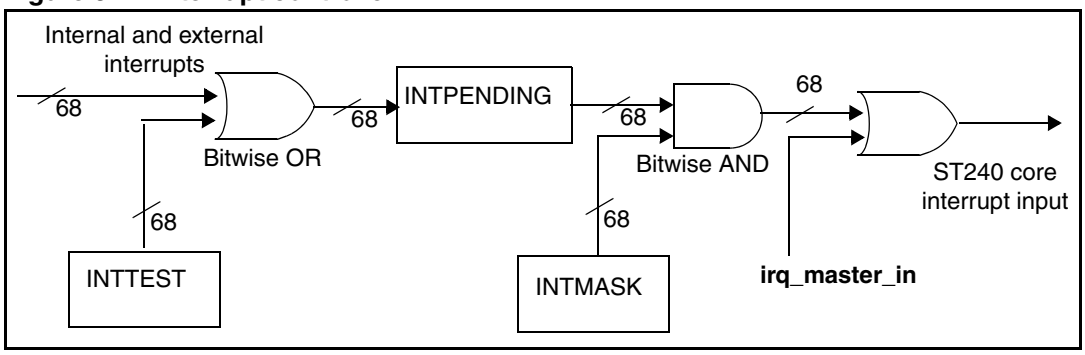
- 64 external interrupts
- three interrupt sources from the timers
- one interrupt source from the performance monitors

Each of these 68 sources has a mask and a test bit associated with it. All interrupts handled by the interrupt controller are normal interrupts as oppose to debug interrupts. See *Section 10.1.1: Interrupt types on page 82* for the definition of the different types.

A non maskable interrupt input (**irq_master_in**) enables the ST240 to use an external interrupt controller.

The structure of the interrupt controller is shown in *Figure 31*.

Figure 31. Interrupt controller



19.1 Operation

An internal or external interrupt causes a pending bit to be set in the interrupt controller and the interrupt to be sent to the core. If interrupts are enabled in the PSW (program status word), the core is interrupted. Every normal interrupt input may be individually masked and asserted by software for test purposes.

There is no hardware support for prioritization of interrupts. It is the responsibility of the software to handle prioritization, assisted by the ST240's ability to mask individual interrupts.

An external interrupt controller can be added to the ST240. When this is the case, all normal interrupts must be masked.

19.2 Interrupt registers

For the addresses of the memory mapped interrupt controller registers see *Chapter 18: Peripheral addresses on page 157*.

The registers are arranged in sets of three. In each set, two registers refer to external interrupts and one refers to internal interrupts.

19.2.1 INTPENDING registers

The three INTPENDING registers show which interrupts are pending.

Table 89. INT_INTPENDING0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	RO	0x0	Interrupt is pending from timer 0
TIMER1	1	RO	0x0	Interrupt is pending from timer 1
TIMER2	2	RO	0x0	Interrupt is pending from timer 2
Reserved	[15:3]	RO	0x0	Reserved
PM_INT	16	RO	0x0	Performance monitoring interrupt is pending
Reserved	[31:17]	RO	0x0	Reserved

Table 90. EXT_INTPENDING0 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_31_0	[31:0]	RO	0x0	External interrupt is pending - system defined, refer to data sheet.

Table 91. EXT_INTPENDING1 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_63_32	[31:0]	RO	0x0	External interrupt is pending - system defined, refer to data sheet.

19.2.2 INTMASK registers

The three INTMASK registers are used to mask all 68 interrupt sources.

Table 92. INT_INTMASK0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	RW	0x0	Mask bit for timer 0
TIMER1	1	RW	0x0	Mask bit for timer 1
TIMER2	2	RW	0x0	Mask bit for timer 2
Reserved	[15:3]	RO	0x0	Reserved
PM_INT	16	RW	0x0	Mask bit for performance monitoring interrupt
Reserved	[31:17]	RO	0x0	Reserved

Table 93. EXT_INTMASK0 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_31_0	[31:0]	RW	0x0	Mask bits for external interrupts - system defined, refer to data sheet.

Table 94. EXT_INTMASK1 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_63_32	[31:0]	RW	0x0	Mask bits for external interrupts - system defined, refer to data sheet.

19.2.3 INTMASKSET and INTMASKCLR registers

These registers support bit-wise access to the INTMASK registers. A store to these locations causes the corresponding bits in the relevant INTMASK register to be cleared or set.

Using this method of accessing the INTMASK registers avoids problems caused by interrupts occurring during a read-modify-write sequence and therefore avoids the need to have interrupts disabled while modifying these registers.

Table 95. INT_INTMASKSET0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	RW	0x0	Mask set bit for timer 0
TIMER1	1	RW	0x0	Mask set bit for timer 1
TIMER2	2	RW	0x0	Mask set bit for timer 2
Reserved	[15:3]	RO	0x0	Reserved
PM_INT	16	RW	0x0	Mask set bit for performance monitoring interrupt
Reserved	[31:17]	RO	0x0	Reserved

Table 96. EXT_INTMASKSET0 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_31_0	[31:0]	RW	0x0	Mask set bits for external interrupts - system defined, refer to data sheet.

Table 97. EXT_INTMASKSET1 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_63_32	[31:0]	RW	0x0	Mask set bits for external interrupts - system defined, refer to data sheet.

Table 98. INT_INTMASKCLR0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	RW	0x0	Mask clear bit for timer 0
TIMER1	1	RW	0x0	Mask clear bit for timer 1
TIMER2	2	RW	0x0	Mask clear bit for timer 2
Reserved	[15:3]	RO	0x0	Reserved
PM_INT	16	RW	0x0	Mask clear bit for performance monitoring interrupt
Reserved	[31:17]	RO	0x0	Reserved

Table 99. EXT_INTMASKCLR0 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_31_0	[31:0]	RW	0x0	Mask clear bits for external interrupts - system defined, refer to data sheet.

Table 100. EXT_INTMASKCLR1 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_63_32	[31:0]	RW	0x0	Mask clear bits for external interrupts - system defined, refer to data sheet.

19.2.4 INTTEST registers

The three INTTEST registers are used to test interrupt inputs.

Table 101. INT_INTTEST0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	RW	0x0	Interrupt test bit for timer 0
TIMER1	1	RW	0x0	Interrupt test bit for timer 1
TIMER2	2	RW	0x0	Interrupt test bit for timer 2
Reserved	[15:3]	RO	0x0	Reserved
PM_INT	16	RW	0x0	Interrupt test bit for performance monitoring interrupt
Reserved	[31:17]	RO	0x0	Reserved

Table 102. EXT_INTTEST0 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_31_0	[31:0]	RW	0x0	Interrupt test bits for external interrupts - system defined, refer to data sheet.

Table 103. EXT_INTTEST1 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_63_32	[31:0]	RW	0x0	Interrupt test bits for external interrupts - system defined, refer to data sheet.

19.2.5 INTTESTSET and INTTESTCLR registers

These registers support bit-wise access to the INTTEST registers. A store to these locations causes the corresponding bits in the relevant INTTEST register to be cleared or set.

Using this method of accessing the INTTEST register avoids the overhead in a direct write of having to read, save and restore unexposed bits.

Table 104. INT_INTTESTSET0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	RW	0x0	Interrupt set bit for timer 0
TIMER1	1	RW	0x0	Interrupt set bit for timer 1
TIMER2	2	RW	0x0	Interrupt set bit for timer 2
Reserved	[15:3]	RO	0x0	Reserved
PM_INT	16	RW	0x0	Interrupt set bit for performance monitoring interrupt
Reserved	[31:17]	RO	0x0	Reserved

Table 105. EXT_INTTESTSET0 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_31_0	[31:0]	RW	0x0	Interrupt set test bits for external interrupts - system defined, refer to data sheet.

Table 106. EXT_INTTESTSET1 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_63_32	[31:0]	RW	0x0	Interrupt set test bits for external interrupts - system defined, refer to data sheet.

Table 107. INT_INTTESTCLR0 bit fields

Name	Bit(s)	Writable	Reset	Comment
TIMER0	0	RW	0x0	Interrupt clear bit for timer 0
TIMER1	1	RW	0x0	Interrupt clear bit for timer 1
TIMER2	2	RW	0x0	Interrupt clear bit for timer 2
Reserved	[15:3]	RO	0x0	Reserved
PM_INT	16	RW	0x0	Interrupt clear bit for performance monitoring interrupt.
Reserved	[31:17]	RO	0x0	Reserved

Table 108. EXT_INTTESTCLR0 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_31_0	[31:0]	RW	0x0	Interrupt clear test bits for external interrupts - system defined, refer to data sheet.

Table 109. EXT_INTTESTCLR1 bit fields

Name	Bit(s)	Writable	Reset	Comment
EXTERN_INT_63_32	[31:0]	RW	0x0	Interrupt clear test bits for external interrupts - system defined, refer to data sheet.

20 Debugging support

User, supervisor and debug modes were introduced in *Chapter 3: Operating modes on page 23*. Debug traps, normal traps, debug breakpoints and normal breakpoints were introduced in *Chapter 10: Traps (exceptions and interrupts) on page 82*.

This chapter further describes the types of breakpoints and the set of debug features which are available on the ST240.

All the debug mode features are available to two kinds of host:

off-chip host	a host connected through the JTAG (test access) port
on-chip host	a host connected on the STBus interface

Debugging support on the ST240 is provided by six main components:

Core	In debug mode the Debug Support Unit (DSU) uses the core to read and write memory and other architectural state. Therefore debug mode is intrusive. The core also processes breakpoints.
Debug support unit	The DSU is the interface between either type of host and the core. Commands to control the core are sent from the host to the DSU using shared Debug Support Registers (DSRs). The core processes commands sent from the host when in debug mode.
Debug ROM	The debug ROM contains the default debug handler which is run after the core enters debug mode. A mechanism is included to install a user defined debug handler. The default debug handler implements a simple set of commands to control the core as described in this chapter.
Debug RAM	1Kbyte of memory mapped RAM. It can only be accessed by the local core when the local core is in debug mode. This is intended for a user defined debug handler.
Host debug interface	Allows access to the DSRs from an off-chip host and supports the sending of debug interrupts to the core.
STBus interface	The STBus interface allows an on-chip host the same functionality as an off-chip host.

Note: All control registers that relate to debug resources are only accessible in debug mode. The names of these registers are prefixed with DBG_ with the exception of SCRATCH4. Access to any such control register in user or supervisor mode raises a CREG_ACCESS_VIOLATION exception.

20.1 Debug resource access

Debug resources have restricted access permissions, so that code running in user or supervisor mode cannot access them. Code running in debug mode has unrestricted access to all resources.

The debug RAM address space has no user or supervisor mode permission. The purpose of the debug RAM address space is to allow the toolchain to install a debug handler which cannot be overwritten by faulty code.

Table 110 shows whether access is permitted by the current operating mode of the ST240. It does not show which loads and stores may access each resource, which is shown in *Table 85: Peripheral access from local core on page 158* or the protected address ranges, which are shown in *Table 84: Peripheral address map on page 157*. There are two levels of checking in this scheme:

- checks in the core against the current operating mode and current permissions (current permissions are only relevant for write access to DSRs)
- checks in the DSU to ascertain whether the access type is suitable for the resource

Table 110. Debug resource access

Debug resource	User mode access	Supervisor mode access	Comment
Control registers with DBG prefix	Denied	Denied	See *Table 76: Control registers - BASE: CREG_BASE on page 146* for full details of access permissions.
Read access to DSRs	Allowed	Allowed	DSRs can always be read.
Write access to DSRs	Denied if DSR_PERMISSIONS[1] is 0 else allowed	Denied if DSR_PERMISSIONS[1:0] is 00 else allowed	Permissions are programmed in DSR_PERMISIONS. Any illegal accesses cause a DTLB DEBUG_VIOLATION exception.
Any access to debug RAM address space	Denied	Denied	Any access (instruction or data) outside debug mode causes a DTLB DEBUG_VIOLATION exception.
Any access to debug ROM address space	Allowed	Allowed	Debug ROM access is unrestricted.

20.1.1 DSR_PERMISSIONS register

DSRs have programmable permissions to allow supervisor code to signal the host using writes to DSR2.

The DSR_PERMISSIONS register controls access to the DSR address space, see *Table 84: Peripheral address map on page 157*. For a description of the DSRs, see *Section 20.3.3: Debug support registers on page 178*.

Table 111. DSR_PERMISSIONS bit fields

Name	Bit(s)	Writable	Reset	Comment
DSR_SUPERVISOR_WRITE_ENABLE	0	RW	0x1	Enables writes to the DSRs and DSR_PERMISSIONS if the core is in supervisor or debug mode.
DSR_USER_WRITE_ENABLE	1	RW	0x0	Enables writes to the DSRs and DSR_PERMISSIONS if the core is in user, supervisor or debug mode.
Reserved	[31:2]	RO	0x0	Reserved.

20.2 Core debugging support

The debug ROM, RAM and the DSRs are discussed further in *Section 20.3: Debug support unit on page 177*.

The DTLB DEBUG_VIOLATION exception is described in *Section 11.4.11: TLB_EXCAUSE register on page 103*, *Figure 25: Instruction access on page 107* and *Figure 26: Data access on page 108*.

20.2.1 Breakpoint support

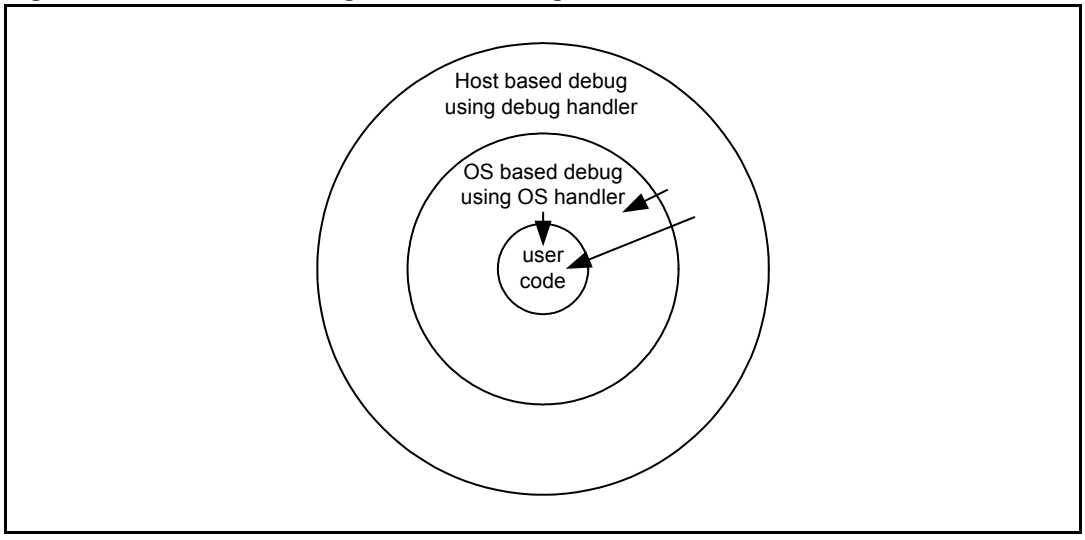
The ST240 provides two classes of breakpoint.

Debug breakpoints — Used for debugging user code or an OS. Debug breakpoints trigger debug exceptions and cause the core to enter debug mode. The handler is described in *Section 20.4.1: Default debug handler on page 181*.

Normal breakpoints — Used for debugging user software under an OS. Normal breakpoints cause normal exceptions. The handler is referenced by TRAP_BREAK, and is conventionally configured by the OS.

Referring to *Figure 32*, the OS based debugger uses normal breakpoints while the host based debugger uses debug breakpoints. Debug breakpoints may be used to debug user code.

Figure 32. Levels of debug what can debug what



This hierarchical scheme allows complete separation of the debugging resources between user code and OS code. Debug mode does not depend upon the OS handlers in any way, and the debug handler is protected from rogue software.

20.2.2 Types of breakpoint

There are three types of normal and debug breakpoint, these are listed in *Table 112*:

Table 112. Summary of breakpoints

Breakpoint type	Debug breakpoints	Normal breakpoints
Software	**dbgsbrk**	**sbrk**
Instruction	DBG_IBREAK	IBREAK
Data	DBG_DBREAK	DBREAK

If a single bundle causes a debug breakpoint and a normal breakpoint of the same type then the debug breakpoint has the highest priority. The full list of priorities is in *Table 34: Trap types and priorities on page 84*.

No debug traps will be raised if the core is in debug mode already. DBG_IBREAK and DBG_DBREAK traps will not be raised if the core is in debug mode and the **dbgsbrk** operation will raise an ILL_INST exception.

20.2.3 Software breakpoints

Software breakpoints are operations that always cause exceptions. The **dbgsbrk** operation either causes a debug exception of type DBG_SBREAK or a normal exception of type ILL_INST. To raise a DBG_SBREAK debug exception the following must be the case:

- the enable bit in DBG_SBREAK_CONTROL must be set, see *Table 113*
- the core must be in user or supervisor mode
- debug mode must not be disabled

  Debug mode is controlled using an input pin and is not visible to the software.

The **sbrk** operation causes a normal exception of type SBREAK and does not have an associated control register.

Software breakpoints allow program execution to halt on a given bundle. Within the trap handler the address which caused the breakpoint is read from SAVED_PC.

Table 113. DBG_SBREAK_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
BRK_ENABLE	0	RW	0x0	Enables **dbgsbrk** instruction. If enabled, **dbgsbrk** causes entry into debug mode. If disabled, it causes an ILL_INST exception.
Reserved	[31:1]	RO	0x0	Reserved.

20.2.4 Hardware breakpoints

Each type of hardware breakpoint is configured by programming:

- control registers to define memory ranges
- control registers to define the behavior and to enable the breakpoints
- enable bits in the PSW for normal breakpoints only; debug breakpoints do not have PSW enable bits

Instruction breakpoints

Instruction breakpoints are configured by programming a range of addresses. These can cause breakpoints to trigger when a bundle, which has a virtual address, is executed:

- within or outside the specified range
- at either limit of the range
- which matches a specified value when masked by another specified value

Any combination of the above cases can be enabled.

Two 32-bit registers are available to define addresses for instruction breakpoints: IBREAK_LOWER and IBREAK_UPPER. Both registers reset to zero. Instruction breakpoints compare the virtual PC of the currently executing bundle against the values programmed in IBREAK_LOWER and IBREAK_UPPER. The comparison ignores bits [1:0] as all bundles are word aligned.

Table 114 lists the bit fields of the control register for the normal instruction breakpoints.

Note: The limit registers are referred to as UPPER and LOWER. This is to allow reuse of the descriptions with the debug breakpoints.

Table 114. IBREAK_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
BRK_IN_RANGE	0	RW	0x0	Break if virtual address is in the inclusive range LOWER to UPPER.
BRK_OUT_RANGE	1	RW	0x0	Break if virtual address is outside the inclusive range LOWER to UPPER.
BRK_EITHER	2	RW	0x0	Break if virtual address is equal to either LOWER or UPPER.
BRK_MASKED	3	RW	0x0	Break if virtual address bitwise ANDed with UPPER equals LOWER.
Reserved	[31:4]	RO	0x0	Reserved.

The debug equivalent of IBREAK is DBG_IBREAK. The control registers are symmetrical with IBREAK registers, and all have a DBG_ prefix. DBG_IBREAK traps are not raised if the core is already in debug mode.

Within the trap handler, the address that has caused the breakpoint is read from SAVED_PC.

Data breakpoints

Data breakpoints are configured by programming a range of addresses. A breakpoint occurs when a data access is made to a virtual address that is:

- within or outside the specified range
- at either limit of the range
- that matches a specified value when masked by another specified value

Any combination of the above cases can be enabled for any combination of the following operation types:

- prefetch
- load
- store

Purges and conditional load/store/prefetches that fail the condition do not trigger data breakpoints. The **stwl** operation always triggers a breakpoint regardless of the state of the lock, see *Chapter 13: Multi-processor and multi-threading support on page 132*.

Two 32-bit registers define addresses for the data breakpoints: DBREAK_LOWER and DBREAK_UPPER. Both reset to zero.

Table 115 lists the bit fields of the control register for normal data breakpoints.

Note: The limit registers are referred to as UPPER and LOWER. This is to allow reuse of the descriptions with the debug breakpoints.

Table 115. DBREAK_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
BRK_IN_RANGE	0	RW	0x0	Break if virtual address is in the inclusive range LOWER to UPPER
BRK_OUT_RANGE	1	RW	0x0	Break if virtual address is outside the inclusive range LOWER to UPPER
BRK_EITHER	2	RW	0x0	Break if virtual address is equal to either LOWER or UPPER
BRK_MASKED	3	RW	0x0	Break if virtual address bitwise ANDed with UPPER equals LOWER
BRK_NOLOAD	4	RW	0x0	Disables triggering on loads from the address region
BRK_NOSTORE	5	RW	0x0	Disables triggering on stores from the address region
BRK_PREFETCH	6	RW	0x0	Enables triggering on prefetches to the address region
Reserved	[31:7]	RO	0x0	Reserved

When a DBREAK exception is raised the virtual address which caused the breakpoint is written to EXADDRESS.

The debug equivalent of DBREAK is DBG_DBREAK. The control registers are symmetrical with DBREAK registers and all have a DBG_ prefix. Therefore the DBG_EXADDRESS register takes the virtual address which caused the breakpoint. DBG_DBREAK traps are not raised if the core is already in debug mode. See *Section 10.7.2: Debug trap startup behavior on page 89* for further details on the DBG_EXADDRESS REGISTER.

20.2.5 Enabling and updating breakpoints

The safe way to enable or update breakpoints is to carry out the following procedure:

1. Write zero to the relevant control register.
2. Program the relevant address registers.
3. Write the required value to the relevant control register.

This sequence prevents spurious breaks due to inconsistent control and address registers.

20.2.6 Branch trace buffer

A control flow change occurs when the ST240 executes a taken conditional branch, an unconditional branch, a procedure call/return, takes a trap or returns from a trap.

The branch trace buffer reports on the last eight control flow changes taken by the ST240, excluding control flow changes in debug mode. The debug trap causing entry into debug mode and the **rfi** causing exit from debug mode are also excluded. The associated control registers are only accessible in debug mode.

For each control flow change the following are recorded:

- physical PC of the bundle causing the control flow change
- physical PC of the control flow change destination bundle

These are recorded in a FIFO queue. The oldest entry is lost if the queue is full when a new control flow change occurs. The sequence of execution can be determined both from this FIFO queue and the original program.

The branch trace buffer is updated when the bundle that caused the change in control flow, or was interrupted, is in writeback.

Reset behavior

When the ST240 is reset:

- the entries in the branch trace buffer are not reset
- the fields in the control register that disable tracing are reset

Therefore following a reset the contents of the branch trace buffer are frozen. This allows the information to be read and used to diagnose which state of the ST240 caused the reset.

Types of control flow change

Three types of control flow changes can be traced selectively if they are taken:

- traps: all normal traps and **rfi** from supervisor mode, no debug traps
- subroutine branch: All **call** and **return** operations
- general branch: **br** (if taken), **brf** (if taken), **goto** operations

The **idle** macro is not traced.

Tracing is enabled or disabled by setting the bits of the DBG_BT_CONTROL register.

Control registers

The branch trace buffer is configured by programming DBG_BT_CONTROL. The bit fields of the DBG_BT_CONTROL register are listed in *Table 116*.

Table 116. DBG_BT_CONTROL bit fields

Name	Bit(s)	Writable	Reset	Comment
BT_TRAP	0	RW	0x0	Enables tracing of normal traps
BT_SUBROUTINE	1	RW	0x0	Enables tracing of subroutine branches
BT_GENERAL	2	RW	0x0	Enables tracing of general branches
Reserved	[7:3]	RO	0x0	Reserved
BT_COUNT	[11:8]	RO	Not reset	Indicates number of valid entries in the branch trace buffer providing at least one control flow change has been traced. Valid values are 0 to 8.
Reserved	[15:12]	RO	0x0	Reserved
BT_RESET	16	RW	0x0	Writing 1 resets all non-reset fields in all branch target buffer registers to 0. Always reads as 0.
Reserved	[31:17]	RO	0x0	Reserved

Use the BT_COUNT field to check how many entries are valid. The contents of invalid trace entries are undefined. The BT_COUNT field is only valid if the branch trace buffer has traced at least one control flow change.

The branch trace buffer control registers are listed in *Table 117*.

Table 117. Branch trace buffer control registers

Name	Bit(s)	Writable	Reset	Comment
DBG_BT_SRC0	[31:0]	RO	Not Reset	Branch trace buffer entry 0 source PC
DBG_BT_DEST0	[31:0]	RO	Not Reset	Branch trace buffer entry 0 destination PC
DBG_BT_SRC1	[31:0]	RO	Not Reset	Branch trace buffer entry 1 source PC
DBG_BT_DEST1	[31:0]	RO	Not Reset	Branch trace buffer entry 1 destination PC
DBG_BT_SRC2	[31:0]	RO	Not Reset	Branch trace buffer entry 2 source PC
DBG_BT_DEST2	[31:0]	RO	Not Reset	Branch trace buffer entry 2 destination PC
DBG_BT_SRC3	[31:0]	RO	Not Reset	Branch trace buffer entry 3 source PC
DBG_BT_DEST3	[31:0]	RO	Not Reset	Branch trace buffer entry 3 destination PC
DBG_BT_SRC4	[31:0]	RO	Not Reset	Branch trace buffer entry 4 source PC
DBG_BT_DEST4	[31:0]	RO	Not Reset	Branch trace buffer entry 4 destination PC
DBG_BT_SRC5	[31:0]	RO	Not Reset	Branch trace buffer entry 5 source PC
DBG_BT_DEST5	[31:0]	RO	Not Reset	Branch trace buffer entry 5 destination PC
DBG_BT_SRC6	[31:0]	RO	Not Reset	Branch trace buffer entry 6 source PC
DBG_BT_DEST6	[31:0]	RO	Not Reset	Branch trace buffer entry 6 destination PC

Table 117. Branch trace buffer control registers (Continued)

Name	Bit(s)	Writable	Reset	Comment
DBG_BT_SRC7	[31:0]	RO	Not Reset	Branch trace buffer entry 7 source PC
DBG_BT_DEST7	[31:0]	RO	Not Reset	Branch trace buffer entry 7 destination PC

The entries are read from the registers in *Table 117*. The higher numbered entries contain the most recent branches.

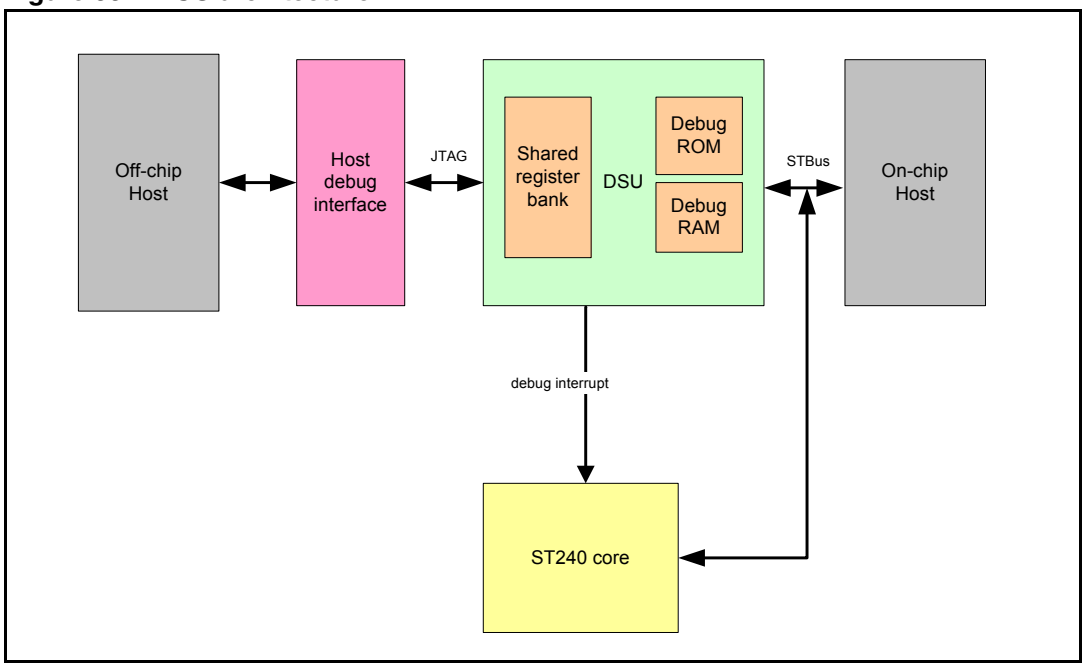
20.3 Debug support unit

The Debug support unit (DSU) enables a host to debug both software and hardware by giving the host direct access to the ST240 core.

20.3.1 Architecture

The architecture of the DSU is shown in *Figure 33*.

Figure 33. DSU architecture



The DSU is controlled by either an on-chip or off-chip host and sends debug interrupts to the core. The core accesses the DSU and the default debug handler code stored in the debug ROM using the STBus. A user defined debug handler can be stored in the debug RAM.

20.3.2 Shared register bank

The bank of 32 shared registers consists of four reserved registers (DSR0 - 3 and DSR31) and 28 general purpose registers (DSR3 - 30). These enable the host to control the core by using the default debug handler software.

The shared register bank is 32-bits wide and the registers are listed in *Table 118*.

Table 118. DSR_REG values

Name	Value	Comment
DSR0	0	DSU version register, contains version number for DSU, core and chip.
DSR1	1	DSU status register, contains DSU control and status bits.
DSR2	2	DSU output register, supports message transfer from ST240 to off-chip host.
DSR3-30	3-30	General purpose registers.
DSR31	31	DSU virtual PC register.

The STBus addresses of the DSU registers are detailed in *Chapter 18: Peripheral addresses on page 157*.

DSR permissions

The DSRs have restricted write permissions to prevent accidental corruption by software running on the local ST240. Unrestricted access is allowed to an on-chip host. Permissions are programmed in DSR_PERMISSIONS.

There are no restrictions on read access to the DSRs. A write which fails the permission check fails silently; an STBus error is not returned. Full details of access permissions are shown in *Table 85: Peripheral access from local core on page 158*.

20.3.3 Debug support registers

The debug support registers (DSR) have restricted write access permissions as defined in *Section 20.1: Debug resource access on page 169*.

DSU version register

The DSU version register (DSR0) is a read-only ID register. The fields are identical to the version register described in *Table 79: VERSION bit fields on page 152*.

DSU status register

The DSU status register (DSR1) contains the DSU status and control bits. The bit fields of DSR1 are listed in *Table 119*.

Table 119. DSR1 bit fields

Name	Bit(s)	Writable	Reset	Comment
DEBUG_INTERRUPT_TAKEN	0	RO	0x0	Value of DEBUG_INTERRUPT_TAKEN signal, active high.
Reserved	[2:1]	RO	0x0	Reserved
BIGENDIAN	3	RO	System defined	1: the core is in big endian mode 0: the core is in little endian mode
HOST_EVENT_ACK_PENDING	4	RO	0x0	The host has received an event command and an event_ack command is pending.
OUTPUT_PENDING	5	RO	0x0	DSR2 contains data to be sent to the off-chip host which has not yet been sent.
TRIGGER_IN	6	RO	0x0	Current value of the trigger in pin.
TRIGGER_OUT	7	RW	0x0	Current value of the trigger out pin.
TRIGGER_ENABLE	8	RW	0x0	Enables/disables debug interrupts on trigger in.
RAISE_DBG_INT	9	RW	0x0	Writing 1 to this bit causes a debug interrupt to be sent to the core. Writing zero has no effect. This bit always reads zero.
APPLY_SOFT_RESET	10	RW	0x0	Writing 1 to this bit causes a soft reset. This bit can only be cleared by a reset.
Reserved	11	RO	0x0	Reserved
IN_IDLE_MODE	12	RO	0x0	This bit is set when the ST240 is in idle mode.
Reserved	[15:13]	RO	0x0	Reserved
SW_FLAGS	[31:16]	RW	0x0	Reserved for future software use.

DSU output register (DSR2)

A value written to DSR2 is sent to the host by an **event** message, which is handshaken with an **event_ack** message, see *DSU to host events on page 190*.

Table 120. DSR2 bit fields

Name	Bit(s)	Writable	Reset	Comment
DATA	[31:0]	RW	0x0	Output data.

20.3.4 Debug support virtual PC register

DSR31 is the debug support virtual PC register (DSVPC). The DSVPC provides visibility of the virtual PC of completed bundles across the debug interface. Not all PC values may be made available by DSVPC and the update of DSVPC is always delayed relative to the actual completion of bundles.

The DSVPC is updated after every bundle completion and is valid once the first bundle has completed following a reset.

This feature allows time based sampling of the virtual PC using the JTAG port or STBus.

20.3.5 Soft reset

As mentioned in *Table 119*, the APPLY_SOFT_RESET bit of DSR1 is used to initiate a soft reset. Soft reset waits for activity to complete before occurring, in a similar way to entering idle mode. The target port does not lose transactions as a result of the reset; all transactions are either completed or blocked until after the reset.

Reset requested by an on-chip host

Soft reset is requested by a writing 1 to the APPLY_SOFT_RESET bit of DSR1 using the STBus. The APPLY_SOFT_RESET bit is sticky and is only cleared by the reset itself. The sequence required to cause the soft reset and wait for completion is:

1. Set APPLY_SOFT_RESET bit of DSR1.
2. Poll DSR1 waiting for the reset value to be read.

Reset requested by an off-chip host

See *Section 20.7: JTAG based host debug interface on page 187* for details of the JTAG commands used in this section.

If the soft reset is requested by the JTAG port, the required sequence is:

1. **poke** DSR1 # write APPLY_SOFT_RESET bit.
2. **nop**.
3. **nop** # receive **poked** DSR1 message, reset now occurs.

The reset is scheduled to occur when the DSU receives the second **nop** message. The host must either have visibility of the reset completed output pin to ensure that the reset sequence has completed before sending any further JTAG messages to the ST240, or it should wait for a sufficiently long time to ensure that the reset has completed. Any messages sent after the second **nop** and before the reset sequence has completed have undefined results.

Soft reset sequence

The actions taken by the core following a soft reset request are:

1. Wait for core to be idle, or 1000 **clk_cpu** cycles, whichever is shorter. The 1000 cycle limit is sufficiently long for all STBus transactions to complete. Waiting for the core to be idle involves:
   - waiting for all STBus transactions to complete
   - waiting for the SDI output buffer to be empty
   - waiting for the pipeline to empty
2. Block access to target port by setting the target port grant to zero and complete all outstanding target port transactions.
3. Reset all SDI master channels.
4. Assert reset to the core and all peripherals. Asserting reset at this stage is identical to asserting normal reset.

20.4 Debug ROM

The debug ROM contains the default debug handler software, which allows simple commands to be sent through the debug interface without requiring a user-defined debug handler.

20.4.1 Default debug handler

A debug trap causes the default debug handler to be executed. If a user defined debug handler has been installed by programming a non zero value into DSR3, the user defined handler is executed; see *Section 20.5: User defined debug handler on page 185*. If DSR3 is zero, the default debug handler waits in the command loop. The default debug handler allocates the usage of DSRs as listed in *Table 121*. This section refers to the registers by name, and not by number.

Table 121. DSU registers as used by the default debug handler

DSR number	Designation	Comment
DSR3	DSR_USER_DEBUG_HANDLER	Control switches to this address if content is non-zero.
DSR4-8	DSU_ARG4-8	Not used in default debug handler.
DSR9	DSU_ARG3	Command argument 3
DSR10	DSU_ARG2	Command argument 2. Used by DSU_POKE and DSU_FLUSH.
DSR11	DSU_ARG1	Command argument 1. Used by all DSU commands.
DSR12	DSU_COMMAND	Command register. Written by host, cleared by ST240 when command accepted.
DSR13	DSU_RESPONSE	Response register. Set by ST240 to a completion code, cleared by host before issuing next command.

Table 121. DSU registers as used by the default debug handler (Continued)

DSR number	Designation	Comment
DSR14	Context saving	Saves R13
DSR15	Context saving	Saves R9
DSR16	Context saving	Saves R10
DSR17	Context saving	Saves R11
DSR18	Context saving	Saves branch bit B0
DSR19	Context saving	Saves R63
DSR20	Context saving	Saves TRAP_EXCEPTION
DSR21	Context saving	Saves SAVED_SAVED_PSW
DSR22	Context saving	Saves SAVED_SAVED_PC
DSR23	Context saving	Saves SAVED_PSW
DSR24	Context saving	Saves SAVED_PC
DSR25	Context saving	Saves EXCAUSENO
DSR26	Context saving	Saves EXADDRESS
DSR27	Context saving	Saves DSR1
DSR28	Context saving	Saves TRAP_TLB
DSR29	Context saving	Saves TRAP_BREAK
DSR30	Context saving	Saves TRAP_INTERRUPT

Command loop

The command loop reads and processes commands from a host to the DSRs. Usage of the designated registers is shown in *Table 122*.

Table 122. Command register usage

Register name	Host use	ST240 use
DSU_COMMAND	Set with command: 1: DSU_CALL_OR_RETURN 2: DSU_FLUSH 3: DSU_POKE 4: DSU_PEEK values greater than four are interpreted as DSU_PEEK	Zeroed when command accepted
DSU_ARG1,2,3	Set with arguments for command, before setting DSU_COMMAND	Set with response arguments before setting DSU_RESPONSE
DSU_RESPONSE	Zeroed after being read	Set to indicate outcome of a command: 1: DSU_PEEKED 2: DSU_POKED 3: DSU_RETURNING 4: DSU_FLUSHED 5: DSU_GOT_EXCEPTION

When the command is complete, the default debug handler stores the results in the argument registers and stores a result code in the DSU_RESPONSE register.

Default handler commands

All commands succeed if no traps occur before completion. If a trap does occur then the default debug handler stores the value DSU_GOT_EXCEPTION into DSU_RESPONSE.

DSU_PEEK — Reads the 32-bit memory location addressed by DSU_ARG1 and returns the data in DSU_ARG1. The address must be word aligned. DSU_RESPONSE is set to DSU_PEEKED.

DSU_POKE — Writes the 32-bit data word in DSU_ARG2 to the memory location addressed by DSU_ARG1. DSU_RESPONSE is set to DSU_POKED.

DSU_CALL_OR_ RETURN — Execute a **call** operation to the user defined routine addressed by DSU_ARG1, or return from the default debug handler if DSU_ARG1 is zero. DSU_RESPONSE is set to DSU_RETURNING in before the **call** or returning from the handler.

This command can be used to call the fast memory transfer routine which is described in *DSU_SAVE_CONTEXT: Context saving on page 185*.

DSU_FLUSH	Purges the address range starting at the value in DSU_ARG1 and ending at the value in DSU_ARG2 from data and instruction caches. DSU_RESPONSE is set to DSU_FLUSHED. Note that the purges are executed with a granularity of 64 bytes.

Calling user defined routines from the default debug handler

As mentioned above the command DSU_CALL_OR_RETURN is used to call a user defined routine or return from the default debug handler. The default debug handler saves the state listed in *Table 121* before calling a user routine. The user routine can overwrite the following registers without the need to save and restore the state:

- registers R9, R10, R11, R13, R63
- branch register B0

The user routine must save and restore any additional required state.

If the user routine does not modify the trap vector registers and a trap occurs during the user routine, the trap handler in the default debug handler is called as described below.

Trap handler with the default debug handler

If a normal trap occurs while a command is being processed, for example, an invalid address supplied to a DSU_PEEK then the trap handler within the default debug handler is called. DSU_RESPONSE is set to DSU_GOT_EXCEPTION**.** The default handler then returns to the command loop. Information about the trap is written to DSRs as shown in *Table 123*.

Table 123. Trap state reporting in the default debug handler

Register	Normal trap
DSU_ARG3	EXADDRESS
DSU_ARG2	EXCAUSENO
DSU_ARG1	SAVED_PC

If a debug trap occurs, the default debug handler will restart.

Context restore and register usage

Before exiting the default debug handler restores any state it has altered, except for SCRATCH4. The default debug handler overwrites SCRATCH4 as temporary storage. Any writes to context saving DSRs during the execution of the default debug handler overwrite the saved state and so should be avoided.

20.5 User defined debug handler

As mentioned above, it is possible to install a user defined debug handler. If DSR3 is non zero, the following sequence causes a jump to the user defined debug handler:

1. Save R9, R10, R11, R13 and R63 as shown in *Table 121*.
2. Save branch bit B0 as shown in *Table 121*.
3. Overwrite SCRATCH4.
4. Replace R13 with the base address of the DSU register block.
5. Execute a **goto** operation to the address in DSR3.

There is no facility for using the state restoration routine in the default debug handler for completion of a user defined debug handler.

20.5.1 Other routines

In addition to the default debug handler the Debug ROM contains the routines listed below to support the operation of a user defined debug handler. The addresses of the routines are shown in *Table 124*. All complete with a **return $r63** operation.

Table 124. Other routines in the debug ROM

Routine	Offset from debug ROM base	Comment
DSU_FMT	0x00c	Fast memory transfer
DSU_FLUSH	0x010	Purge buffer from caches
DSU_SAVE_CONTEXT	0x014	Save context
DSU_RESTORE_CONTEXT	0x018	Restore context

DSU_SAVE_CONTEXT: Context saving

This routine saves all registers and branch registers into a section of memory in the debug RAM.

Table 125. Context saving in the debug RAM

Offset into debug RAM	Data stored
0x28 to 0x10c	R1 to R62
0x110	All branch registers. Bits [3:0] save B0,..., bits [31:28] save B7

DSU_FMT: Fast memory transfer

DSU_FMT provides more efficient access to memory than sending DSU_PEEK and DSU_POKE commands. DSU_FMT overwrites the following state:

- R9, R10, R11, R13, B0
- DSR3 to 30

Calling DSU_FMT causes a second command loop to be run, which supports the states shown in *Table 127*. DSR29 stores the command state.

Table 126. Command register usage for DSU_FMT

Register name	Host use	ST240 use
DSR3	Number of words to transfer (1 to 24)	None
DSR4	Base address of the access	None
DSR5 to 28	Data for transfer	Data for transfer
DSR29	Set with command: 15: READ_CMD 11: WRITE_CMD 10: EXIT_CMD	Set to indicate status: 12: WAIT_FOR_COMMAND 13: WRITE_BUF_ON_GOING 14: READ_BUF_ON_GOING 1638: EXITED (0x666)
DSR30	Saves R3	

The meaning of each command state is shown in *Table 127*.

Table 127. Command states for DSU_FMT

State	Entered by	Comment
READ_CMD	Host	The host requests a read command
WRITE_CMD	Host	The host requests a write command
EXIT_CMD	Host	The hosts requests to exit the command loop
WAIT_FOR_COMMAND	ST240	The command loop is waiting for a command. All commands have finished.
WRITE_BUF_ON_GOING	ST240	A multiple poke command is running
READ_BUF_ONGOING	ST240	A multiple peek command is running

Sequence to run a multiple poke command

The host polls DSR29 until it sees WAIT_FOR_COMMAND. Then the host writes to DSR3 to DSR28 as defined above and finally writes WRITE_CMD to DSR29 to start the command. The ST240 updates DSR29 to WRITE_BUF_ON_GOING during the command then updates DSR29 to WAIT_FOR_COMMAND.

Sequence to run a multiple peek command

The host polls DSR29 until it sees WAIT_FOR_COMMAND. Then the host writes to DSR3 and DSR4 as defined above and finally writes READ_CMD to DSR29 to start the command. The ST240 updates DSR29 to READ_BUF_ON_GOING during the command then to WAIT_FOR_COMMAND. The data is available in DSR5 to DSR28.

Sequence to exit the command loop

The host polls DSR29 until it sees WAIT_FOR_COMMAND. Then the host writes EXIT_CMD to DSR29 to exit the command loop. The ST240 updates DSR9 and DSR29 to EXITED and executes a **return** operation to R63. The host polls for the EXITED state.

DSU_FLUSH

This routine takes the same arguments as DSU_FLUSH as defined in *Section 20.4.1: Default debug handler on page 181*. The only difference is that when called in this way the routine executes a **return** operation to R63 upon completion instead of returning to the default debug handler command loop.

DSU_RESTORE_CONTEXT: Context saving

This routine restores all registers and branch registers from a section of memory in the debug RAM as defined in *Table 125 on page 185*.

20.6 Debug RAM

The ST240 core contains a 1Kbyte Debug RAM. This RAM is memory mapped, but can only be accessed by the local ST240 in debug mode. The purpose of the RAM is to allow installation of a user defined debug handler, which is protected from corruption by code running in user or supervisor mode.

The Debug RAM address space is accessible from the host using the memory access commands available in the default debug handler.

20.7 JTAG based host debug interface

Exchange of information with the off-chip host is through the host debug interface as shown in *Figure 33: DSU architecture on page 177*. All references to the host in this section refer to the off-chip host.

All JTAG based communication is done with **peek**, **poke**, **peeked**, **poked**, **nop**, **event** and **event_ack** commands exchanged between the host and the DSU.

The JTAG interface provides access only to the registers within the DSU, these registers are described in *Section 20.3.2: Shared register bank on page 178*.

Access is made to memory via a software convention with the core as described in *Section 20.4.1: Default debug handler on page 181*.

event messages can be sent in either direction to allow software running on the core to synchronize with software on the host.

20.7.1 Protocol and flow control

40-bit commands are sent between the host and the DSU through the JTAG port. Whenever a command is sent to the DSU by the host, the DSU responds with a response from a previous command, or a **nop** if no response is pending.

A symmetrical protocol is employed where every action request is handshaken. Therefore:

- a **peek** from the host is acknowledged with a **peeked**
- a **poke** from the host is acknowledged with a **poked**
- an **event** from the host is acknowledged with an **event_ack**
- an **event_ack** from the host does not require a response and invokes a **nop** in reply if no other response is pending
- a **nop** from the host does not require a response and invokes a **nop** if no other response is pending

The DSU sends a response to the command *i* either after the DSU receives command (*i*+1), or after it receives command (*i*+2).

In its initial state, the DSU responds to command *i* after it receives command (*i*+1) and continues to do so until the processor writes to DSR2, sending an **event** to the host. The sending of the **event** is prioritized over the sending of a response to command *i* which is buffered.

In the state where there is a buffered response, DSU responds to command *i* after it receives command (*i*+2). When the DSU receives an **event_ack** or a **nop** as command (*j*+2), it sends the response to command *(j*+1) and then re-enters the initial state as the buffer is now empty since neither the **event_ack** nor the **nop** require a response.

As only one **event** can be outstanding to the host at once, the DSU is only required to buffer one response. As responses are not always immediately sent to incoming commands the host must account for every **peek** and **poke** which is sent. The host must also poll the DSU with **nop**s to receive **event**s.

20.7.2 Command format

Commands supported across the JTAG interface are as listed in *Table 128*. Commands are 40 bits long and consist of an 8-bit header and a 32-bit data field. The header is split into two fields. Commands are sent over the JTAG interface bit[0] first.

Table 128. JTAG commands

Command	header[2:0]	header[7:3]	data[39:8]	Command needs a response	Action / comment
Commands from the Host to the DSU					
nop	0x0	0x0	0x0(1)	No	No action. No command is currently waiting to be sent. **nop**s can be used to poll for **event**s.
peek	0x1	Address	0x0(1)	Yes	Request to peek the DSU register specified by the address field. The DSU replies with **peeked, address, value**(2).
poke	0x2	Address	Data	Yes	Request to poke a DSU register specified by the address field with the value specified by the data field. The DSU replies with **poked, address, 0**(1).
event	0x3	0x0(3)	reason[10:8] channel[13:11] 0x0[39:14]	Yes	If reason = 1, channel = 0 raise a debug interrupt, otherwise a debug interrupt is not raised. The DSU replies with **event_ack, reason, channel, 0**(1).
event_ack	0x4	0x0(1)	DSR2[31:0]	No	An event from the DSU to the host has been processed. The original word in DSR2 is returned, but is not used.
reserved(4)	0x5-0x7	Undefined	Undefined	No	Reserved commands are treated as **nop**s.
Commands from the DSU to the host					
nop	0x0	0x0	0x0	No	No command is currently waiting to be sent.
peeked	0x1	Address	Value	No	Peeked data being returned to the host.
poked	0x2	Address	0x0	No	Response to a request to poke a DSU register.
event	0x3	0x0	DSR2[31:0]	Yes	DSR2 [31:0] is copied into the data field and the use is software defined. Must be eventually replied to by **event_ack**.

Table 128. JTAG commands (Continued)

Command	header[2:0]	header[7:3]	data[39:8]	Command needs a response	Action / comment
event_ack	0x4	0x0	reason[10:8] channel[13:11] 0x0[39:14]	No	An event has been processed (a debug interrupt has been applied to the core, it may not have been processed yet). The data field from the incoming command is placed in the data field of the response command.
reserved(4)	0x5-0x7	Undefined	Undefined	No	The behavior is defined by the host software.

1. The implementation does not check the value of this field
2. The response may be delayed by one message if an **event_ack** is outstanding, as described in *Section 20.7.1: Protocol and flow control on page 188*.
3. The implementation checks the value of this field
4. Commands marked reserved are for future development.

20.7.3 Handling events

Event messages can be sent in either direction (host to DSU and DSU to host) to allow software running on the core to synchronize with software on the host. This section describes how event messages are handled by the DSU and the host.

Host to DSU events

The DSU generates an **event_ack** in response to an **event** command. If reason = 1 and channel = 0 in the event command, a debug interrupt is raised. The response to a debug interrupt indicates only that it has been raised, not that it has been taken. The host can determine whether the ST240 is in debug mode by peeking DSR1. Bit 0 is set on entering debug mode and is cleared on exiting.

If the ST240 has not returned from debug mode, a subsequent event command causes an additional debug interrupt. The controlling software must ensure that events are not sent until previous events have been completely processed.

DSU to host events

Multiple events can be sent from the host to the DSU, but only one outstanding DSU to host event is permitted.

Two bits in DSR1 (OUTPUT_PENDING DSR1[5] and HOST_EVENT_ACK_PENDING DSR1[4]) provide information about the current DSU to host event status; this is illustrated in *Table 129*.

Table 129. Status of events and DSR1 bit fields

DSR1[5]	DSR1[4]	Comment
0	0	No outstanding DSU to host **event.**
1	0	DSR2 has been written to, **event** has not been sent yet. Writes to DSR2 before DSR1[4] is set do not cause extra events, but update the value of DSR2 which is sent with the **event**.
1	1	This case does not occur, DSR1[5] and DSR1[4] are mutually exclusive.
0	1	The **event** has been sent, writes to DSR2 do not cause further events to be sent.
0	0	The **event_ack** has been received, writes to DSR2 cause **event** again.

20.8 On-chip host debug interface

The ST240 supports debugging from an on-chip host over the STBus, by reading and writing the DSRs. There are only two significant changes for the operation of an on-chip host.

- Debug interrupts are raised by writing to the RAISE_DBG_INT bit of DSR1, instead of sending an **event** message.
- Writes to DSR2 do not cause an **event** message to be sent to the on-chip host. A replacement mechanism must be constructed in software.

The default debug handler processes exactly the same commands from either type of host. The only difference is whether the DSRs are accessed through the JTAG port or the STBus.

20.9 Non software controllable behavior

Two operations are provided that depend upon input pins to the ST240. These are:

- causing the ST240 to come out of reset into idle mode and to wait for a debug interrupt
- disabling of debug mode on a production system

21 Performance monitoring

The ST240 provides a hardware instrumentation system. This system is accessible by several control registers:

- a register which controls the performance monitoring (PM_CR)
- a 64-bit core clock counter (PM_PCLK, PM_PCLKH)
- four event counters (PM_CNT*i*, *i* = 0, 1, 2, 3)
- an event interrupt control (PM_INT)

Of the control registers listed above only PM_PCLK and PM_PCLKH have read permission in user mode. None have write permission in user mode.

The system allows software to simultaneously monitor up to four of the predefined events listed in *Table 130*.

21.1 Events

The programmable events supported by the ST240 are listed in *Table 130*. All reserved counters always read zero.

Table 130. PM_EVENT values

Name	Value	Comment
PM_EVENT_DHIT	0	Number of cached loads and stores that hit the data cache.
PM_EVENT_DMISS	1	Number of cached loads and stores that miss the data cache.
PM_EVENT_DSTALLCYCLES	2	Number of cycles the core is stalled waiting for load/store operations to complete These include DTLB and uncached stalls, but excludes instruction cache purges, div/rem operations and stalls due to DTCM access from the target port.
PM_EVENT_PFTISSUED	3	Number of prefetches that are sent to the STBus.
PM_EVENT_PFTHITS	4	Number of cached loads that hit the prefetch cache.
PM_EVENT_WBHITS	5	The number of writes that hit the write buffer.
PM_EVENT_IHIT	6	Number of instruction cache hits.
PM_EVENT_IMISS	7	Number of instruction cache misses.
PM_EVENT_IMISSCYCLES	8	Number of cycles the instruction cache was stalled for waiting for the STBus.
PM_EVENT_IFETCHSTALL	9	Number of cycles of instruction fetch stall.
PM_EVENT_BUNDLES	10	Number of bundles completed.
PM_EVENT_LDST	11	Number of load/store class operations executed (includes data cache purges, syncs).

Table 130. PM_EVENT values (Continued)

Name	Value	Comment
PM_EVENT_TAKENBR	12	Number of taken branches (includes **br**, **brf**, **return**, **rfi**, **goto** and **call**).
PM_EVENT_NOTTAKENBR	13	Number of not taken branches (**br** and **brf**).
PM_EVENT_EXCEPTIONS	14	Number of exceptions and debug traps taken.
PM_EVENT_INTERRUPTS	15	Number of normal interrupts taken.
PM_EVENT_BUSREADS	16	Number of read transactions issued to the STBus (this is the number of uncached reads, instruction and data cache refills and prefetches issued to the STBus).
PM_EVENT_BUSWRITES	17	Number of write transactions issued to the STBus (this is the number of write buffer lines evicted and the number of uncached writes issued to the STBus).
PM_EVENT_OPERATIONS	18	Number of completed operations.
PM_EVENT_WBMISSES	19	Number of cached writes that missed the data cache and missed the write buffer (this excludes cache line evictions).
PM_EVENT_BUBBLEORNOP	20	Number of completed bundles that contained only nops plus the number of pipeline bubbles. Only counts **nop**s encoded as the nop macro and not, for example, **ldwc**, which fails the condition check. Bubbles are caused by fetch stalls, branch stalls and interlock stalls.
PM_EVENT_LONGIMM	21	Number of immediates extensions in completed bundles.
PM_EVENT_ITLBMISS	22	Number of instruction cache reads that missed the ITLB.
PM_EVENT_DTLBMISS	23	Number of load/store operations that missed the DTLB when the TLB is enabled. Includes instruction cache purges that use the TLB.
PM_EVENT_UTLBHIT	24	Number of accesses to the UTLB that were hits.
PM_EVENT_ITLBWAITCYCLES	25	Number of cycles the instruction cache spends waiting for the ITLB to fill.
PM_EVENT_DTLBWAITCYCLES	26	Number of cycles the data cache spends waiting for the DTLB to fill.
PM_EVENT_UTLBARBITRATION CYCLES	27	Number of cycles where the ITLB or DTLB was waiting for access to the UTLB because the UTLB was busy servicing a request.
Reserved	28	Reserved.
PM_EVENT_PFTEVICTIONS	29	Number of times a newly issued prefetch has evicted an existing prefetch cache entry.
PM_EVENT_IDLECYCLES	30	Number of cycles the ST240 remained in idle mode.

Table 130. PM_EVENT values (Continued)

Name	Value	Comment
Reserved	31	Reserved.
PM_EVENT_DTCMARBITRATIONCYCLES	32	Number of cycles that a DTCM access from the target port had to wait until it gained access to the DTCM.
PM_EVENT_DTCMACCESSES	33	Number of accesses made to the DTCM from the target port.
PM_EVENT_DTCMSTALLCYCLES	34	Number of cycles where access from the target port to the DTCM caused the data cache to be stalled.
PM_EVENT_DTCMHITS	35	Number of cached loads or stores accessing the DTCM.
Reserved	36-39	Reserved.
PM_EVENT_NORMALTRAP	40	Number of normal traps taken.
PM_EVENT_TRAPEXCEPTION	41	Number of traps taken using TRAP_EXCEPTION.
PM_EVENT_TRAPINTERRUPT	42	Number of traps taken using TRAP_INTERRUPT.
PM_EVENT_TRAPBREAK	43	Number of traps taken using TRAP_BREAK.
PM_EVENT_TRAPTLB	44	Number of traps taken using TRAP_TLB.
PM_EVENT_DEBUGTRAP	45	Number of debug traps taken.
Reserved	46-49	Reserved.
PM_EVENT_DFILLS	50	Number of cached loads which missed the cache and the prefetch cache and so caused an STBus request.
PM_EVENT_DFILLCYCLES	51	Number of cycles spent waiting for the STBus due to cached loads missing the data cache and the prefetch cache.
PM_EVENT_DUNCACHEDLOADS	52	Number of uncached loads issued to the STBus.
PM_EVENT_DUNCACHEDLOADCYCLES	53	Number of cycles spent waiting for the STBus due to uncached loads.
Reserved	54-59	Reserved.
PM_EVENT_WAITLCYCLES	60	Number of cycles spent idle due to a **waitl** operation waiting for the ATOMIC_LOCK to be free.
PM_EVENT_STWLSUCCESSES	61	Number of times a **stwl** operation reached writeback when the lock bit was set. Only incremented if the **stwl** did not raise an exception.
PM_EVENT_STWLFAILS	62	Number of times a **stwl** operation reached writeback when the lock bit was not set. Only incremented if the **stwl** did not raise an exception.
PM_EVENT_DIVREMSTALLCYCLES	63	Number of cycles spent stalled due to executing a divide or remainder operation.
Reserved	64-127	Reserved.

The time delay between an event occurring and the count being recorded in the relevant register is implementation dependent.

21.2 Control register (PM_CR)

The ST240 uses this control register (PM_CR) to reset and enable all the counters and define the events of the four programmable count registers. The control register's bit fields are listed in *Table 131*.

Table 131. PM_CR bit fields

Name	Bit(s)	Writable	Reset	Comment
ENB	0	RW	0x0	1: counting is enabled 0: counting is disabled This applies to PM_CNT0-3, PM_PCLK and PM_PCLKH.
RST	1	RW	0x0	When a 1 is written to this field all the counters (PM_CNT0-3, PM_PCLK and PM_PCLKH) are reset to 0. If a 0 is written it is ignored. This field does not retain its value and so always reads as 0.
IDLE	2	RW	0x0	Indicates whether the core has been in idle mode. When the core enters idle mode this bit is set to 1. Writing a 0 to this bit has no effect. Writing a 1 to this bit clears the bit.
Reserved	3	RO	0x0	Reserved.
EVENT0	[10:4]	RW	0x0	7-bit field specifying the event being monitored for this counter.
EVENT1	[17:11]	RW	0x0	7-bit field specifying the event being monitored for this counter.
EVENT2	[24:18]	RW	0x0	7-bit field specifying the event being monitored for this counter.
EVENT3	[31:25]	RW	0x0	7-bit field specifying the event being monitored for this counter.

21.3 Event counters (PM_CNT*i*)

Each of the four programmable event counter (PM_CNT*i*, *i* = 0 to 3) is incremented each time the specified countable event occurs. The event counters can record any one of the events specified in *Table 130 on page 192*.

Reading from an event counter register returns the current event count. Writing to an event counter changes the current count. If a counter is enabled, read or written at the same time as an event triggers the counter to increment, the increment is ignored.

21.4 64bit clock counter (PM_PCLK, PM_PCLKH)

The two 32-bit registers PM_PCLK and PM_PCLKH together form a 64-bit counter. PM_PCLK contains the lower 32 bits, PM_PCLKH contains the upper 32 bits.

It is not possible to read both registers in a single, atomic operation, as 64-bit loads are not supported to control registers. The counter must be read using two load word operations. It must be checked that the values returned by the two loads are consistent as the sequence may have been interrupted.

A suitable procedure for reading the counter is provided in this code fragment:

```
long long read_counter(void) {
  unsigned int high1;
  unsigned int low;
  unsigned int high2;
  do {
    high1 = *(volatile int *)PM_PCLKH;
    low = *(volatile int *)PM_PCLK;
    high2 = *(volatile int *)PM_PCLKH;
  }
  while (high1 != high2);

  return (((long long) high1) << 32) | ((long long) low);
}
```

21.5 Recording events

To start recording, write the desired fields to an ST240 general purpose register. This can be achieved by first reading the PM_CR register, then modifying the fields as appropriate.

The ENB bit must be set to 1. The RST bit must be set to 1, if the counters are to be reset. The four programmable counter fields (EVENT*i* (*i* = 0 to 3) of the PM_CR register) must be modified to the value representing the events to be counted, see the value column of *Table 130 on page 192*.

The value in the register is then written to the memory mapped PM_CR for the operation to begin.

To stop recording, read the value of PM_CR, set the ENB bit to zero, then write back the result to PM_CR. Do not change any other bits. If the RST bit is set to 1 then the PM_CNT*i* registers are reset.

Whilst counting events over a long period of time, the 32-bit counters may overflow (the exception to this is PM_PCLK, which has been extended to 64-bits as described in *Section 21.4*). To obtain a continuous profile, it is recommended that interrupts are enabled, see *Section 21.6*. If event interrupts are not enabled, the counters will overflow silently.

21.6 Interrupts generated by performance monitors

Event counters may be set up to generate interrupts upon overflow. Interrupt generation is controlled by the PM_INT control register. The bit fields of the PM_INT register are described in *Table 132*.

Table 132. PM_INT bit fields

Name	Bit(s)	Writable	Reset	Comment
ENABLE0	0	RW	0x0	Interrupt mask for event 0.
ENABLE1	1	RW	0x0	Interrupt mask for event 1.
ENABLE2	2	RW	0x0	Interrupt mask for event 2.
ENABLE3	3	RW	0x0	Interrupt mask for event 3.
Reserved	[7:4]	RO	0x0	Reserved.
PENDING0	8	RW	0x0	When 1 an interrupt is pending for event 0. Writing a 0 to this bit has no effect. Writing 1 to this bit clears the bit. When PM_CNT0 overflows and the value wraps this bit is set to 1.
PENDING1	9	RW	0x0	As for PENDING0, but for PM_CNT1.
PENDING2	10	RW	0x0	As for PENDING0, but for PM_CNT2.
PENDING3	11	RW	0x0	As for PENDING0, but for PM_CNT3.
Reserved	[31:12]	RO	0x0	Reserved

The value of each of the PENDING[0-3] bits is set when the associated event counter wraps. The ENABLE[0-3] bits are masks for the four interrupt sources.

The PM_INT interrupt in the interrupt controller (see *Chapter 19: Interrupt controller on page 163*) will be raised if one of the following is true:

- PENDING0 and ENABLE0 are set
- PENDING1 and ENABLE1 are set
- PENDING2 and ENABLE2 are set
- PENDING3 and ENABLE3 are set

Three usage models are suggested.

- For a complete count of an event that may overflow, the counter is reset to zero and the interrupt is used to record the number of overflows.
- To count *n* events the counter may be set to *-n*. It will then cause an interrupt after *n* events.
- The PENDING[0-3] bits are used to extend the counters to 33 bits as they indicate overflow. In this mode all ENABLE[0-3] bits are zero if interrupts are not required.

Note: Interrupts cannot be triggered by PM_PCLK or PM_PCLKH overflowing.

21.7 PM counters in idle mode

When in idle mode, only the PM_EVENT_IDLECYCLES counter increments. No other counters will change state.

21.8 STBus latency measurement

The performance monitors can be used to measure the latency of STBus transactions.The latency of the STBus for instruction cache fills, data cache fills and uncached loads can be calculated as follows:

PM_EVENT_IMISSCYCLES / PM_EVENT_IMISS

PM_EVENT_DFILLCYCLES / PM_EVENT_DFILLS

PM_EVENT_DUNCACHEDLOADCYCLES / PM_EVENT_DUNCACHEDLOADS

22 Execution model

This chapter describes how bundles are executed in terms of their component operations.

In the absence of traps, the core fetches a bundle from memory, decodes the operations within it and reads their operands. It then executes the operations in parallel and writes the results back to the architectural state of the machine. All operations in a bundle commit their results to the state of the machine at the same point in time; this is known as the commit point.

In the presence of traps, the core uses the commit point to distinguish between recoverable and non-recoverable traps.

Traps that are detected prior to the commit point are treated as recoverable. They are recoverable because the machine state has not been updated, which means that the state prior to the execution of the bundle can be recovered. In some cases, the cause of the trap can be corrected and the bundle restarted.

Traps detected after the commit point are non-recoverable. The machine state has been updated and in some cases it may not be clear which bundle caused the trap. Non-recoverable traps are consequently of a serious nature and cannot be restarted. On the ST240, the only class of non-recoverable trap is an STBus error resulting from a store operation.

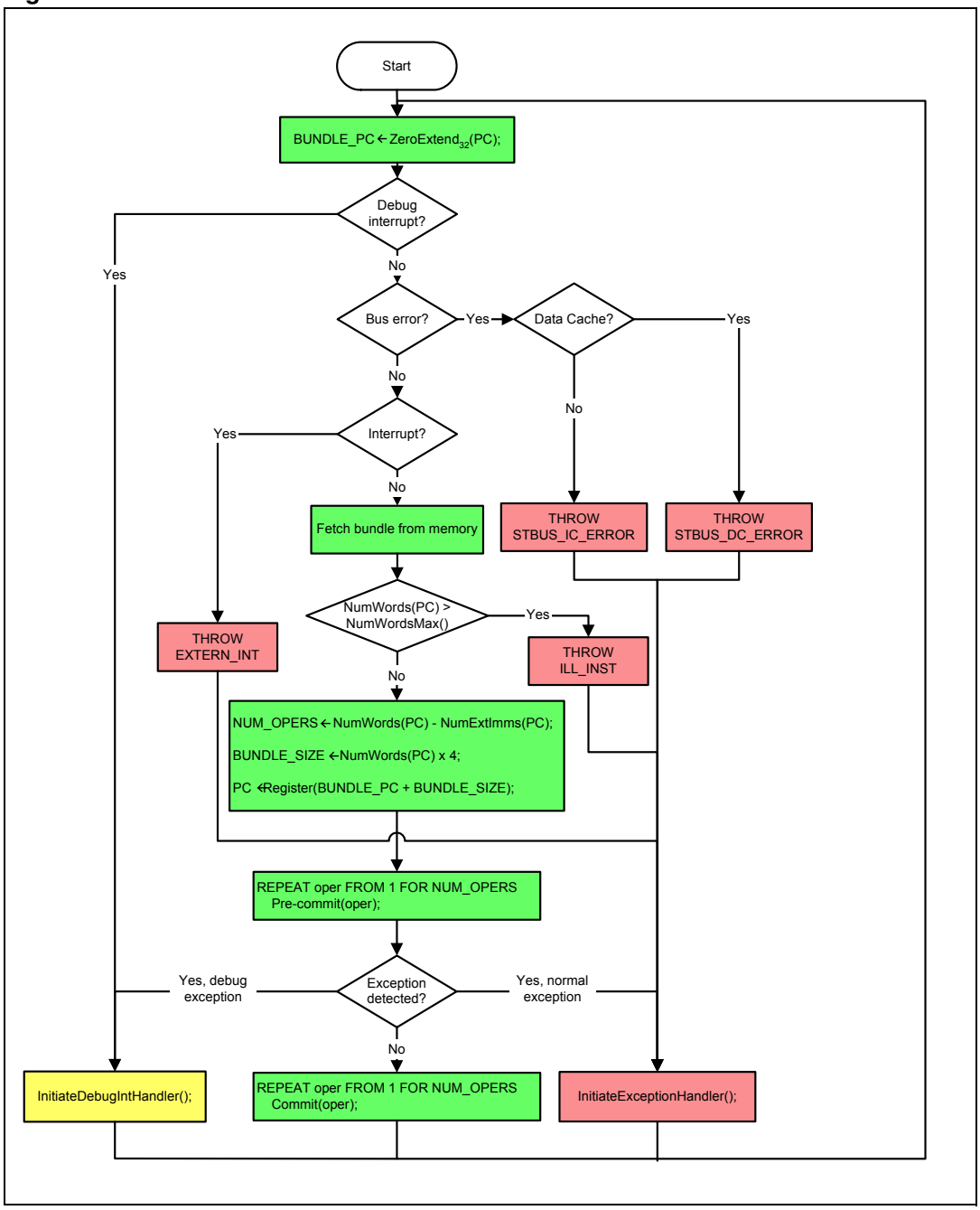
22.1 Bundle fetch, decode, and execute

The ST240 specifies the fetch, decode and execution of bundles using an abstract sequential model to show the effects on the architectural state of the machine. In this abstract model, each bundle is executed sequentially with respect to other bundles. This means that the core completes all actions associated with a specific bundle before starting any action associated with the following bundle.

Specific implementations of the ST240 are generally designed to deliver substantial optimizations on the scheme provided by this abstract model. However, for legal bundle sequences that permit execution latency, these optimizations are not visible architecturally. The behavior in illegal cases is defined by the *Chapter 10: Traps (exceptions and interrupts) on page 82*. Operation latencies are described in *Chapter 6: Execution pipeline and latencies on page 33*.

The execution flow shown in *Figure 34* uses the notation defined in *Chapter 23: Specification notation on page 202*. There are additional functions available that can be used to extract details from bundles; these are described in *Section 22.2: Functions on page 201*.

Figure 34. Execution model



22.2 Functions

The flow chart in *Figure 34* includes a number of functions. Those functions are described in this section.

22.2.1 Bundle decode

The ST240 uses the functions listed in *Table 133* in the bundle decode phase.

Table 133. Bundle decode functions

Function	Description
`NumWords(address)`	Returns the number of words in the bundle starting at `address`. The return value is equal to the number of contiguous words, starting from `address`, without their stop bit set +1.
`NumExtImms(address)`	Returns the number of extended immediates in the bundle starting at `address`.
`NumWordsMax()`	Returns the maximum number of words in a bundle. This is equivalent to the issue width of the core.

22.2.2 Operation execution

The ST240 uses the functions listed in *Table 134* in the operation execution phase.

Table 134. Operation execution functions

Function	Description
`Pre-commit(n)`	For the n^{th} operation in the bundle, execute the pre-commit phase, see *Section 24.2: Operation specifications on page 223*(1).
`Commit(n)`	For the n^{th} operation in the bundle, execute the commit phase, see *Section 24.2: Operation specifications on page 223*(1).

1. `n` is a value in the range [1... number of operations in the bundle].

22.2.3 Exceptional cases

The ST240 uses the functions listed in *Table 135* in exceptional cases.

Table 135. Operation execution functions

Function	Description
`InitiateExceptionHandler()`	Execute the statements defined in *Section 10.7: Saving and restoring execution state on page 87*.
`InitiateDebugIntHandler()`	Execute the statements defined in *Section 20.2.1: Breakpoint support on page 171*.

23 Specification notation

This chapter describes the formal language used in this manual for describing operations, exceptions and interrupts. The language has the following features:

- a simple variable and type system, see *Section 23.1*
- expressions, see *Section 23.2*
- statements, see *Section 23.3*
- notation for the architectural state of the machine, see *Section 23.4*

Additional mechanisms are defined to model memory (*Section 23.5.2*), for control registers (*Section 23.5.3*), and cache instructions (*Section 23.5.4*).

Chapter 24: Instruction set describes each instruction using informal text as well as the formal language. Occasionally it is not appropriate for one of these descriptions to describe the full semantics of the instruction; in such cases, both descriptions must be taken into account to constitute the full specification. In the case of an ambiguity or a conflict, the notational language takes precedence over the text.

Note: This chapter does not currently contain all the functions listed in the instruction pages.

23.1 Variables and types

Variables are used to hold state. The type of a variable determines the set of values that the variable can hold and the operators that can be applied to it. The scalar types are integers, booleans and bit fields. One-dimensional arrays of scalar types are also supported.

The architectural state of the machine is represented by a set of variables. Each of these variables has an associated type, which is either a bit field or an array of bit fields. Bit fields provide a bit-accurate representation of the variables.

The core uses additional variables to hold temporary values. The type of a temporary variable is determined by its context rather than explicit declaration. The type of a temporary variable can be an integer, a boolean or an array of integers or booleans.

23.1.1 Integer

An integer variable can take the value of any mathematical integer. No limits are imposed on the range of integers supported. Integers obey their standard mathematical properties. Integer operations do not overflow. The integer operators are defined so that singularities cannot occur. For example, no definition is given to the result of divide by zero; the operator is simply not available when the divisor is zero.

The representation of literal integer values is achieved using the following notations:

- unsigned decimal numbers are represented by the regular expression: [0-9]+
- signed decimal numbers are represented by the regular expression: -[0-9]+
- hexadecimal numbers are represented by the regular expression: 0x[0-9a-fA-F]+
- binary numbers are represented by the regular expression: 0b[0-1]+

These notations are standard and map onto integer values in the obvious way. Underscore characters (‘_’) can be inserted into any of the above literal representations to aid readability. They do not change the represented value.

23.1.2 Boolean

A boolean variable can take two values.

- Boolean false. The literal representation of boolean false is *FAL*SE.
- Boolean true. The literal representation of boolean true is *TR*UE.

23.1.3 Bit fields

Bit fields are provided to define ‘bit-accurate’ storage.

Bit fields containing arbitrary numbers of bits are supported. A bit field of b bits contains bits numbered from 0 (the least significant bit) up to b-1 (the most significant bit). Each bit can take the value 0 or the value 1.

Bit fields are mapped to, and from, unsigned integers in the conventional way. If bit i of a b-bit bit field, where i is in [0, b), is set then it contributes 2^i to the integral value of the bit field. The integral value of the bit field as a whole is an integer in the range [0, 2b).

Bit fields are mapped to, and from, signed integers using two’s complement representation. This is as above, except that the bit b-1 of a b-bit bit field contributes -2(b-1) to the integral value of the bit field. The integral value of the bit field as a whole is an integer in the range [-2b-1, 2b-1].

A bit field may be used in place of an integer value. In this case the integral value of the bit field is used. A bit field variable may be used in place of an integer variable as the target of an assignment. In this case the integer must be in the range of values supported by the bit field.

23.1.4 Arrays

One-dimensional arrays of the above types are also available. Indexing into an n-element array A is achieved using the notation A[i] where A is an array of some type and i is an integer in the range [0, n). This selects the ith. element of the array A. If i is zero this selects the first entry, and if i is n-1 then this selects the last entry. The type of the selected element is the base type of the array.

Multi-dimensional arrays are not provided.

23.2 Expressions

Expressions are constructed from monadic operators, dyadic operators and functions applied to variables and literal values.

There are no defined precedence and associativity rules for the operators. Parentheses are used to specify the expression unambiguously.

Sub-expressions can be evaluated in any order. If a particular evaluation order is required, then sub-expressions must be split into separate statements.

23.2.1 Integer arithmetic operators

For arithmetic, the notation uses common mathematical operators. The standard dyadic operators are listed in *Table 136*.

Table 136. Standard dyadic operators

Operation	Description
i + j	Integer addition
i - j	Integer subtraction
i × j	Integer multiplication
i / j	Integer division(1)
i \ j	Integer remainder(1)

1. These operators are defined only where **j <> 0**

The division operator truncates towards zero. The remainder operator is consistent with this. The sign of the result of the remainder operator follows the sign of the dividend. Division and remainder are not defined for a divisor of zero.

For a numerator (**n**) and a denominator (**d**), the following properties apply where d≠0:

$$n = d \times (n / d) + (n \backslash d)$$
$$(-n) / d = -(n / d) = n / (-d)$$
$$(-n) \backslash d = -(n \backslash d)$$
$$n \backslash (-d) = n \backslash d$$
$$0 \leq (n \backslash d) < d \quad \text{where} \quad n \geq 0 \quad \text{and} \quad d > 0$$

The standard monadic operators are described in *Table 137*.

Table 137. Standard monadic operators

Operator	Description
- i	Integer negation
|i|	Integer modulus (absolute value)

23.2.2 Integer shift operators

The available integer shift operators are listed in *Table 138*.

Table 138. Shift operators

Operation	Description
n << b	Integer left shift
n >> b	Integer right shift

The shift operators are defined on integers as follows where b ≥ 0:

$$n \ll b = n \times 2^{b}$$

$$n \gg b = \begin{cases} n / 2^{b} & \text{where} \quad n \geq 0 \\ (n - 2^{b} + 1) / 2^{b} & \text{where} \quad n < 0 \end{cases}$$

Note that right shifting by **b** places is a division by $\mathbf{2^b}$ with the result rounded towards minus infinity. This contrasts with division, which rounds towards zero, and is the reason why there are separate right shift definitions for positive and negative **n**.

23.2.3 Integer bitwise operators

The available integer bitwise operators are listed in *Table 139*.

Table 139. Bitwise operators

Operation	Description
i ∧ j	Integer bitwise AND
i ∨ j	Integer bitwise OR
i ⊕ j	Integer bitwise XOR
~ i	Integer bitwise NOT
$n_{<b \text{ FOR } m>}$	Integer bit field extraction: extract m bits starting at bit b from integer n
$n_{<b>}$	Integer bit field extraction: extract 1 bit starting at bit b from integer n

In order to define bitwise operations, all integers are considered as having an infinitely long two's complement representation. Bit 0 is the least significant bit of this representation, bit 1 is the next higher bit, and so on. The value of bit b, where b ≥ 0, in integer n is given by:

$$\text{BIT}(n, b) = (n / 2^{b}) \backslash 2 \quad \text{where} \quad n \geq 0$$

$$\text{BIT}(n, b) = 1 - \text{BIT}((-n - 1), b) \quad \text{where} \quad n < 0$$

Care must be taken whenever the infinitely long two's complement representation of a negative number is constructed. This representation contains an infinite number of higher bits with the value 1 representing the sign. Typically, a subsequent conversion operation is used to discard these upper bits and return the result back to a finite value.

Bitwise AND ($\wedge$), OR ($\vee$), XOR ($\oplus$) and NOT ($\neg$) are defined on integers as follows, where b takes all values such that $b \geq 0$:

$$BIT(i \wedge j,\ b) = BIT(i,\ b) \times BIT(j,\ b)$$
$$BIT(i \vee\ j,\ b) = BIT(i \wedge j,\ b) + BIT(i \oplus j,\ b)$$
$$BIT(i \oplus j,\ b) = (BIT(i,\ b) + BIT(j,\ b))\backslash 2$$
$$BIT(\sim i,\ b) = 1 - BIT(i,\ b)$$

Note: Bitwise NOT of any finite positive i results in a value containing an infinite number of higher bits with the value 1 representing the sign.

Bitwise extraction is defined on integers as follows, where $b \geq 0$ and $m > 0$:

$$n\langle\ b \text{ FOR } m\rangle = (n \gg b) \wedge (2^m - 1)$$
$$n\langle\ b\rangle = n\langle\ b \text{ FOR } 1\rangle$$

The result of $n_{<b \text{ FOR } m>}$ is an integer in the range [0, 2m).

23.2.4 Relational operators

Relational operators compare integral values and give a boolean result.

Table 140. Relational operators

Operation	Description
i = j	Result is *TRUE* if i is equal to j, otherwise *FALSE*
i ≠ j	Result is *TRUE* if i is not equal to j, otherwise *FALSE*
i < j	Result is *TRUE* if i is less than j, otherwise *FALSE*
i > j	Result is *TRUE* if i is greater than j, otherwise *FALSE*
i ≤ j	Result is *TRUE* if i is less than or equal to j, otherwise *FALSE*
i ≥ j	Result is *TRUE* if i is greater than or equal to j, otherwise *FALSE*

23.2.5 Boolean operators

Boolean operators perform logical AND, OR, XOR and NOT. These operators have boolean sources and result. Additionally, the conversion operator INT is defined to convert a boolean source into an integer result.

Table 141. Boolean operators

Operation	Description
i AND j	Result is *TRUE* if i and j are both true, otherwise *FALSE*
i OR j	Result is *TRUE* if either/both i and j are true, otherwise *FALSE*
i XOR j	Result is *TRUE* if exactly one of i and j are true, otherwise *FALSE*
NOT i	Result is *TRUE* if i is false, otherwise *FALSE*
INT i	Result is 0 if i is false, otherwise 1

23.2.6 Single-value functions

In some cases, it is inconvenient or inappropriate to describe an expression directly in the specification language. In these cases a function call is used to reference the undescribed behavior.

A single-value function evaluates to a single value (the result), which can be used in an expression. The type of the result value can be determined by the expression context from which the function is called. There are also multiple-value functions which evaluate to multiple values. These are only available in an assignment context, and are described in *Section 23.3.2: Assignment on page 210*.

Functions may generate side-effects.

Arithmetic functions

Table 142. Arithmetic functions

Function	Description
`CountLeadingZeros(i)`	Convert integer **i** to 32-bit bit field and return the number of leading zeros in the bit field. For example: If $i_{<31>}$ is 1 then the return value is 0. If all bits are 0 then the return value is 32.

Scalar conversions

Two monadic functions are defined to support conversion from integers to bit-limited signed and unsigned number ranges. For a bit-limited integer representation containing **n** bits, the signed number range is $[-2^{n-1}, 2^{n-1}]$ while the unsigned number range is $[0, 2^n]$.

These functions are often used to convert between signed and unsigned bit-limited integers and between bit fields and integer values.

Table 143. Integer conversion operators

Function	Description
$\texttt{ZeroExtend}_n\texttt{(i)}$	Convert integer i to an n-bit 2's complement unsigned range
$\texttt{SignExtend}_n\texttt{(i)}$	Convert integer i to an n-bit 2's complement signed range

These two functions are defined as follows, where $n > 0$:

$$\text{ZeroExtend}_n(i) = i_{\langle\ 0\ \text{FOR}\ n\rangle}$$

$$\text{SignExtend}_n(i) = \begin{cases} i_{\langle\ 0\ \text{FOR}\ n\rangle} & \text{where} \quad i_{\langle\ n-1\rangle} = 0 \\ i_{\langle\ 0\ \text{FOR}\ (n-1)\rangle} - 2^n & \text{where} \quad i_{\langle\ n-1\rangle} = 1 \end{cases}$$

For syntactic convenience, conversion functions are also defined for converting an integer or boolean to a single bit and to a value which can be stored as a 32-bit register. *Table 144* shows the additional functions provided.

Table 144. Conversion operators from integers to bit fields

Operation	Description
`Bit(i)`	If i is a boolean, this is equivalent to Bit(INT i) Otherwise, convert lowest bit of integer i to a 1-bit value This is a convenient notation for $i_{<0>}$
`BranchRegister(i)`	If i is a boolean, this is equivalent to **BranchRegister(INT i)** Otherwise, convert lowest 4 bits of integer **i** to an unsigned 4-bit value. This is a convenient notation for $i_{<0\ FOR\ 4>}$
`Register(i)`	If i is a boolean, this is equivalent to **Register(INT i)** Otherwise, convert lowest 32 bits of integer **i** to an unsigned 32-bit value. This is a convenient notation for $i_{<0\ FOR\ 32>}$

Logical functions

The logical functions provided by the ST240 are listed in *Table 145*.

Table 145. Logical functions

Function	Description
`MaskAndShiftn(i,j)`	Returns ($ZeroExtend_n(i)$)<<(j*n)
`UnsignedExtractn(i,j)`	Returns $ZeroExtend_n$(i>>(j*n))
`SignedExtractn(i,j)`	Returns $SignExtend_n$(i>>(j*n))

Saturating functions

The saturating functions provided by the ST240 are listed in *Table 146*.

Table 146. Saturating functions

Function	Description
$Saturate_n$`(i)`	Clamp the value of `i` to an n bit signed integer and sign extend.
$UnsignedSaturate_n$`(i)`	Clamp the value of `i` to an n bit signed integer and zero extend.
$Overflow_n$`(i)`	Returns true if the value of `i` cannot be represented by an n bit signed integer.

Floating point functions

The behavior of floating point operations is described in *Section 7.8: Floating point operations on page 51*. The floating point functions provided by the ST240 are listed in *Table 147*.

Table 147. Floating point functions

Function	Description
`FAddSNonIeee(i,j)`	Single precision floating point IEEE format addition
`FSubSNonIeee(i,j)`	Single precision floating point IEEE format subtraction
`FCompEqNonIeee(i,j)`	Single precision floating point IEEE format equality comparison
`FCompGENonIeee(i,j)`	Single precision floating point IEEE format greater than or equal to comparison
`FCompGtNonIeee(i,j)`	Single precision floating point IEEE format greater than comparison
`FMulSNonIeee(i,j)`	Single precision floating point IEEE format multiplication
`IntToSFloatNonIeee(i)`	Signed integer to single precision floating point IEEE format conversion
`SFloatToIntNonIeee(i)`	Single precision floating point IEEE format to signed integer conversion

Divide and remainder functions

The behavior of divide and remainder operations is described in *Section 7.10: Divide and remainder operations on page 57*. The divide and remainder functions provided by the ST240 are listed in *Table 148*.

Table 148. Divide and remainder functions

Function	Description
`IDivIeee(i,j)`	Signed integer divide
`IRemIeee(i,j)`	Signed integer remainder
`UIDivIeee(i,j)`	Unsigned integer divide
`UIRemIeee(i,j)`	Unsigned integer remainder

23.3 Statements

An instruction specification consists of a sequence of statements. These statements are processed sequentially in order to specify the effect of the instruction on the architectural state of the machine. The available statements are discussed in this section.

Each statement is terminated with a semi-colon. A sequence of statements can be aggregated into a statement block using '**{**' to introduce the block and '**}**' to terminate the block. A statement block can be used anywhere that a statement can.

23.3.1 Undefined behavior

The statement:

```
UNDEFINED();
```

indicates that the resultant behavior is architecturally undefined.

A particular implementation can choose to specify an implementation-defined behavior in such cases. It is very likely that implementation-defined behavior will vary from implementation to implementation. Exploitation of implementation-defined behavior should be avoided to allow software to be portable between implementations.

In cases where architecturally undefined behavior can occur in user mode, the implementation ensures that implemented behavior does not break the protection model. Thus, the implemented behavior is some execution flow that is permitted for that user mode thread.

23.3.2 Assignment

The notation uses the '←' operator to denote assignment of an expression to a variable. An example assignment statement is:

```
variable ← expression;
```

The expression can be constructed from variables, literals, operators and functions as described in *Section 23.2: Expressions on page 203*. The expression is fully evaluated before the assignment takes place. The variable can be an integer, a boolean, a bit field or an array of any one of these types.

Assignment to architectural state

This is where the variable is part of the architectural state, as described in *Table 149: Scalar architectural state on page 213*. The type of the expression and the type of the variable must match, or the type of the variable must be able to represent all possible values of the expression.

Assignment to a temporary variable

Alternatively, if the variable is not part of the architectural state, then it is a temporary variable. The type of the variable is determined by the type of expression. A temporary variable must be assigned to, before it is used in the instruction specification.

Assignment of an undefined value

An assignment of the following form results in a variable being initialized with an architecturally undefined value:

```
variable ← UNDEFINED;
```

After assignment the variable holds a value that is valid for its type. However, the value is architecturally undefined. The actual value can be unpredictable; that is to say the value indicated by `UNDEFINED` can vary with each use of `UNDEFINED`. Architecturally-undefined values can occur in both user and privileged modes.

A particular implementation can choose to specify an implementation-defined value in such cases. It is very likely that any implementation-defined values will vary from implementation to implementation. If software is intended to be portable between ST240 implementations, then exploitation of implementation-defined values should be avoided.

Assignment of multiple values

Multi-value functions return multiple values, and are only available when used in a multiple assignment context. The syntax of a multiple assignment consists of a list of comma-separated variables, an assignment symbol followed by a function call. The function is evaluated and returns multiple results into the variables listed. The number of variables and the number of results of the function must match. The assigned variables must all be distinct, that is, no aliases.

For example, a two-valued assignment from a function call with three parameters can be represented as:

```
variable1, variable2 ← call(param1, param2, param3);
```

23.3.3 Conditional

Conditional behavior is specified using the keywords `IF`, `ELSE IF` and `ELSE`.

Conditions are expressions that result in a boolean value. If the condition after an `IF` is true, then its block of statements is executed and the whole conditional is considered complete, ignoring any `ELSE IF` or `ELSE` clauses, if they exist. If the condition is false, then each `ELSE IF` clauses are processed, in turn, in the same manner. If no conditions are met and an `ELSE` clause exists, then its statement block is executed. Finally, if no conditions are met and there is no `ELSE` clause, then the statement has no effect apart from the evaluation of the condition expressions.

The `ELSE IF` and `ELSE` clauses are optional. In ambiguous cases, the `ELSE` matches with the preceeding `IF` or `ELSE IF`.

For example:

```
IF (condition1)
  block1
ELSE IF (condition2)
  block2
ELSE
  block3
```

23.3.4 Repetition

Repetitive behavior is specified with the following construct:

```
REPEAT i FROM m FOR n STEP s
  block
```

The block of statements is iterated `n` times, with the integer `i` taking the values:

m, m +s, m + 2s, m + 3s, up to m + (n - 1)s.

The behavior is equivalent to textually writing the block `n` times with `i` being substituted with the appropriate value in each copy of the block.

The value of `n` must be greater or equal to `0`, and the value of `s` must be non-zero. The values of the expressions for `m`, `n` and `s` must be constant across the iteration. The integer **`i`** must not be assigned with a new value within the iterated block. The `STEP` clause can be omitted, in which case the step-size takes the default value of 1.

23.3.5 Exceptions

Exception handling is triggered by a `THROW` statement (for normal exceptions) or a `DbgThrow()` procedure call (for debug exceptions). When an exception is thrown, no further statements are executed from the operation specification; no architectural state is updated. Furthermore, if any one of the operations in a bundle triggers an exception, none of the operations in that bundle update the architectural state.

If any operation in a bundle triggers an exception then the exception is taken. The actions associated with the taking of an exception are described in *Section 10.7.1: Normal trap startup behavior on page 87* and *Section 10.7.2: Debug trap startup behavior on page 89*.

There are two forms of throw statement:

```
THROW type;
```

and:

```
THROW type, value;
```

where `type` indicates the type of exception that is launched, and `value` is an optional argument to the exception handling sequence. If `value` is not given, then it is undefined.

There is only one form of `DbgThrow()`:

```
DbgThrow(type);
```

The exception types and priorities are described in *Chapter 10: Traps (exceptions and interrupts) on page 82*.

23.3.6 Procedures

Procedure statements contain a procedure name followed by a list of comma-separated arguments contained within parentheses and followed by a semi-colon. The execution of procedures typically causes side-effects to the architectural state of the machine.

Procedures are generally used when it is difficult or inappropriate to specify the effect of an instruction using the abstract execution model. A fuller description of the effect of the instruction is given in the surrounding text.

An example procedure with two parameters is:

```
proc(param1, param2);
```

23.4 Architectural state

Chapter 4: Architectural state on page 27 contains a full description of the visible state of the ST240. The notations used in the specification to refer to this state are summarized in *Table 149* (for scalar variables) and *Table 150* (for arrays). Each item of scalar architectural state is a bit field of a particular width. Each item of array architectural state is an array of bit fields of a particular width.

Table 149. Scalar architectural state

Architectural state	Type is a bit field containing:	Description
PC	32 bits	Program counter. During bundle execution PC points to the next bundle in the instruction stream as shown in *Figure 34: Execution model on page 200*.
PSW	32 bits	Program status word.
SAVED_PC	32 bits	Copy of the PC used during interrupts.
SAVED_PSW	32 bits	Copy of the PSW used during interrupts.
SAVED_SAVED_PC	32 bits	Copy of the PC used during debug interrupts.
SAVED_SAVED_PSW	32 bits	Copy of the PSW used during debug interrupts
R*i* where *i* is in [0, 63]	32 bits	64 x 32-bit general purpose registers. R0 reads as zero. Assignments to R0 are ignored.
P*i* where *i* is even and within the range [0, 60]	64 bits	31 x 64 bit general purpose registers, which access the 32-bit general purpose registers in pairs. P0 reads 64 bits of zero. P62 is forbidden due to link register restrictions.
LR	32 bits	Link register, synonym for R63.
B*i* where *i* is in [0, 7]	4 bit	8 x 4-bit branch registers.

Table 150. Array architectural state

Architectural state	Type is an array of bit-fields each containing:	Description
CR*i* where *i* is index of the control register	32 bits	Control registers, for which some specifications refer to individual control registers by their names, as defined in the *Chapter 15: Control registers on page 145*.
MEM[i] where *i* is in [0, 2^{32}]	8 bits	2^{32} bytes of memory.

23.5 Memory and control registers

This section describes the additional formal language defined to model memory (*Section 23.5.2*), for control registers (*Section 23.5.3*), and cache instructions (*Section 23.5.4*).

23.5.1 Support functions

The functions used in the memory and control register descriptions are listed in *Table 151*.

Table 151. Support functions

Function	Description
`Misalignedn(address)`	Result is *TRUE* if `address` is not n-bit aligned, otherwise *FALSE*.
`NoTranslation(address)`	Result is *TRUE* if the TLB is enabled and has no mapping for `address`, otherwise *FALSE*.
`MultiMapping(address)`	Results is *TRUE* if the TLB has more than one mapping for `address`, otherwise *FALSE*.
`Translate(address)`	Looks up `address` in the TLB and returns the associated physical address.
`SCUHit(paddress)`	Results is *TRUE* if the physical address, `paddress`, hit the SCU, otherwise *FALSE*.
`ReadAccessViolation(address)`	Result is *TRUE* if the TLB is enabled and a read access to `address` is not permitted by the TLB, otherwise *FALSE*.
`WriteAccessViolation(address)`	Result is *TRUE* if the TLB is enabled and a write access to `address` is not permitted by the TLB, otherwise *FALSE*.
`IsCRegSpace(address)`	Result is *TRUE* if `address` is in the control register space, otherwise *FALSE*.
`UndefinedCReg(address)`	Result is *TRUE* if `address` does not correspond to a defined control register, otherwise *FALSE*.
`CRegIndex(address)`	Returns the index of the control register which maps to `address`.
`CRegReadAccessViolation(index)`	Result is *TRUE* if read access is not permitted to the given control register, otherwise *FALSE*.
`CRegWriteAccessViolation(index)`	Result is *TRUE* if write access is not permitted to given control register, otherwise *FALSE*.
`BusReadError(paddress)`	Result is *TRUE* if reading from physical address, `paddress`, generates a Bus Error, otherwise *FALSE*.
`IsDBreakHit(address)`	Result is *TRUE* if `address` triggers a data breakpoint, otherwise it is *FALSE*.
`IsDBreakPrefetchHit(address)`	Result is *TRUE* if `address` triggers a prefetch breakpoint, otherwise it is *FALSE*.

23.5.2 Memory model

The instruction specification uses a simple model of memory access to define the relationship between the content of a logical memory and the values manipulated by instructions. The simple model ignores any caches that may be present; their operation is defined by the text of the architecture manual.

The processor's view of logical memory is defined in terms of an array MEM[i] defined in *Table 150 on page 213*. The mapping between the logical memory and a physical memory is described in *STBus endian behavior on page 494*.

The notation MEM[s FOR n] is used to denote an 8*n-bit bit field created from the concatenation of the n elements MEM[s] through MEM[s+i-1], where i (the byte number) varies in the range [0, n). The value of MEM[s FOR n] depends on the endianness of the processor.

- If the processor is operating in little endian mode then:

$$(\mathrm{MEM}[s \text{ FOR } n])\langle 8i \text{ FOR } 8\rangle = \mathrm{MEM}[s+i]$$

  This equivalence states that byte number i in the bit field MEM[s FOR n] is the ith. byte in memory counting upwards from MEM[s].

- If the processor is operating in big endian mode then:

$$(\mathrm{MEM}[s \text{ FOR } n])\langle 8i \text{ FOR } 8\rangle = \mathrm{MEM}[(s+n-1)-i]$$

  This equivalence states that byte number i, using big endian byte numbering (that is, byte 0 is bits 8n-8 to 8n-1), in the bit field MEM[s FOR n] is the ith. byte in memory counting downwards from MEM[n].

For syntactic convenience, functions and procedures are provided to read and write memory.

Support functions

The specification of the memory instructions relies on the support functions listed in *Table 151: Support functions on page 214*. These functions are used to model the behavior of the TLB described in *Chapter 11: Memory translation and protection on page 93*.

Reading memory

The following functions are provided to support the reading of memory:

Table 152. Memory read functions

Function	Description
`ReadCheckMemory`$_n$`(address)`	Throws any non-Bus Error exception generated by an n-bit read from `address`
`PrefetchCheckMemory(address)`	Throws any BusError exceptions generated by a prefetch from `address`
`ReadMemory`$_n$`(address)`	Issues an n-bit read to `address` (can generate BusError exception)
`ReadMemResponse()`	Returns the value of the read request issued

The ReadCheckMemory$_n$ procedure takes an integer parameter to indicate the address being accessed. The number of bits being read (n) is either 8, 16, or 32. The procedure throws any alignment or access violation exceptions generated by a read access to that address.

```
ReadCheckMemory_n(a);
```

is equivalent to:

```
IF (Misalignedn(a))
   THROW MISALIGNED_TRAP, a;
IF (PSW[TLB_ENABLE])
   IF (NoTranslation(a) OR
      MultiMapping(a) OR
      ReadAccessViolation(a))
      THROW DTLB, a;
```

The ReadMemory$_n$ procedure takes an integer parameter to indicate the address being accessed. The number of bits being read (*n*) is either 8, 16, or 32. The required bytes are read from memory, interpreted according to endianness, and the read bit field value assigned to a temporary integer. If the read memory value is to be interpreted as signed, then use a sign-extension when accessing the result using ReadMemResponse. The procedure call:

```
ReadMemory_n(a);
```

is equivalent to:

```
pa = Translate(a);
width ← n / 8;
IF (BusReadError(pa))
   THROW BUS_DC_ERROR, a; // Non-recoverable
mem_response ←MEM[pa FOR width];
```

The function ReadMemResponse returns the data that has been read from memory. The assignment:

```
result ← ReadMemResponse();
```

is equivalent to:

```
result ← mem_response;
```

Prefetching memory

The following procedure is provided to denote memory prefetch.

Table 153. Memory prefetch procedure

Function	Description
`PrefetchMemory(address)`	Prefetch memory if possible

This is used for a software-directed data prefetch from a specified effective address. This is a hint to give advance notice that particular data will be required. `PrefetchMemory`, performs the implementation-specific prefetch when the address is valid:

```
PrefetchMemory(a);
```

This is equivalent to:

```
IF (NOT NoTranslation(a) AND
  NOT MultiMapping(a) AND
  NOT ReadAccessViolation(a)) {
    pa = Translate(a);
    IF (SCUHit(pa))
      Prefetch(a);
  }
```

where `Prefetch` is a cache operation defined in *Section 23.5.4: Cache model on page 219*. Prefetching memory does not generate any exceptions.

Writing memory

The procedures listed in *Table 154* are provided to write memory.

Table 154. Memory write procedures

Function	Description
`WriteCheckMemoryn(address)`	Throws any exception generated by an n-bit write to `address`
`WriteMemoryn(address, value)`	Aligned n-bit write to memory

The $\texttt{WriteCheckMemory}_n$ procedure takes an integer parameter to indicate the address being accessed. The number of bits being written (n) is either 8, 16, or 32. The procedure throws any alignment or access violation exceptions generated by a write access to that address.

$\texttt{WriteCheckMemory}_n\texttt{(a);}$

is equivalent to:

```
IF (Misaligned_n(a))
  THROW MISALIGNED_TRAP, a;
IF (NoTranslation(a) OR
  MultiMapping(a) OR
  WriteAccessViolation(a))
  THROW DTLB, a;
```

The $\texttt{WriteMemory}_n$ procedure takes an integer parameter to indicate the address being accessed, followed by an integer parameter containing the value to be written. The number of bits being written (n) is either 8, 16, 32 or 64 bits. The written value is interpreted as a bit field of the required size; all higher bits of the value are discarded. The bytes are written to memory, ordered according to endianness. The statement:

```
WriteMemory_n(a, value);
```

is equivalent to:

```
pa = Translate(a);
width ←n / 8;
MEM[pa FOR width] ←value<0 FOR n>;
```

23.5.3 Control register model

This section describes the procedures for reading and writing to control registers.

Reading control registers

The following procedures are provided for reading from control registers.

Note: Only word (32-bit) control register accesses are supported.

Table 155. Control register read functions

Function	Description
`ReadCheckCReg(address)`	Throws any exception generated by reading from `address` in the control register space
`ReadCReg(address)`	Issues a read from the control register mapped to `address`

The `ReadCheckCReg` procedure takes an integer parameter to indicate the address being accessed. The procedure throws any alignment or non-mapping exception generated by reading from the control register space.

```
ReadCheckCReg(a);
```

is equivalent to:

```
IF (UndefinedCReg(a))
  THROW CREG_NO_MAPPING, a;
index ←CRegIndex(a);
IF (CRegReadAccessViolation(index))
  THROW CREG_ACCESS_VIOLATION, a;
```

The control register file is denoted CR. The function `ReadCReg` is provided:

```
ReadCReg(a);
```

is equivalent to:

```
index ←CRegIndex(a);
mem_response ←CR_index;
```

Writing control registers

The following procedures are provided for writing to control registers.

Note: Only word (32-bit) control register accesses are supported.

Table 156. Control registers write procedures

Function	Description
`WriteCheckCReg(address)`	Throws any exception generated by writing to the `address` in the control register space
`WriteCReg(address, value)`	Writes `value` to the control register mapped to `address`

The `WriteCheckCReg` procedure takes an integer parameter to indicate the address being accessed. The procedure throws any alignment, non-mapping or access violation exceptions generated by writing to the control register space:

```
WriteCheckCReg(a);
```

This is equivalent to:

```
IF (UndefinedCReg(a))
  THROW CREG_NO_MAPPING, a;
index ←CRegIndex(a);
IF (CRegWriteAccessViolation(index))
  THROW CREG_ACCESS_VIOLATION, a;
```

A procedure called WRITECREG is provided to write control registers:

```
WriteCReg(a, value);
```

is equivalent to:

```
index ←CRegIndex(a);
CR_index ←value;
```

23.5.4 Cache model

Cache operations are used to prefetch and purge lines in caches. The effects of these operations are beyond the scope of the specification language, and are therefore modelled using procedure calls. The behavior of these procedure calls is elaborated in the *Chapter 12: Memory subsystem on page 110*.

Table 157. Procedures to model cache operations

Procedure	Description
`PurgeInsAddress()`	Purge address from the L1 instruction cache and any L2 cache, see *Purging the L1 instruction cache by address on page 116*.
`PurgeInsAddressL1()`	Purge address from the L1 instruction and not from any L2 cache, see *Purging the L1 instruction cache by address on page 116*.
`PurgeInsSet()`	Purge a set of lines from the L1 instruction cache and the corresponding section of any L2 cache, see *Purging a memory range from the data cache on page 123*.

Table 157. Procedures to model cache operations (Continued)

Procedure	Description
`PurgeInsSetL1()`	Purge a set of lines from the L1 instruction cache excluding any L2 cache, see *Purging a memory range from the data cache on page 123*.
`Sync()`	Data memory subsystem synchronization function, see *Section 12.5.3: Memory ordering on page 122*).
`PurgeInsAddressCheckMemory(address)`	Throws any exceptions generated by purging addresses from the instruction cache, see *Purging the L1 instruction cache by address on page 116*.
`PurgeAddressCheckMemory(address)`	Throws any exceptions generated by purging addresses from the data cache, see *Purging data by address on page 122*.
`PurgeAddress(address)`	Purge address from the L1 data cache and any L2 cache, see *Purging data by address on page 122*.
`PurgeAddressL1(address)`	Purge address from the L1 data cache and not from any L2 cache, see *Purging data by address on page 122*.
`FlushAddress(address)`	Flush address from the L1 data cache and any L2 cache, see*Flushing data by address on page 123*.
`FlushAddressL1(address)`	Flush address from the L1 data cache and not from any L2 cache, see *Flushing data by address on page 123*.
`InvalidateAddress(address)`	Invalidate address from the L1 data cache and any L2 cache, see*Invalidating data by address on page 123*.
`InvalidateAddressL1(address)`	Invalidate address from the L1 data cache and not from any L2 cache, see *Invalidating data by address on page 123*.
`PurgeSet(address)`	Purge a set of lines from the L1 data cache and a corresponding section of any L2 cache, see *Purging the L1 instruction cache and L2 cache by set on page 117*.
`PurgeSetL1 (address)`	Purge a set of lines from the L1 data cache and not from any L2 cache, see *Purging the L1 instruction cache and L2 cache by set on page 117*.
`Prefetch(address)`	Prefetch a data cache line if it is in cacheable memory, see *Prefetching data on page 121*.

Table 157. Procedures to model cache operations (Continued)

Procedure	Description
`Wmb()`	Write memory barrier. See *Table 70: Summary of operation behavior for MP, MT and UP on page 138* for the function.
`Dib()`	Data/instruction barrier. See *Table 70: Summary of operation behavior for MP, MT and UP on page 138* for the function.

23.5.5 Architectural state model

Architectural state such as the PC and PSW is modified by a number of procedures. These procedures also have the effect of flushing the pipeline; this is beyond the scope of the specification language.

Table 158. Procedures to model changing architectural state

Procedure	Description
`Rfi()`	Return From Interrupt. This flushes the pipeline, see *Section 10.7.3: Restoring execution state on page 90*.
`PswMask(value1,value2)`	PSW <- (PSW & (~value2)) | (value1 & value2). This flushes the pipeline, see *Section 4.4.2: PSW access on page 29*.

23.5.6 Other functions

The functions in this section do not fit into the other categories listed in this chapter.

Table 159. Procedures to model changing architectural state

Procedure	Description
`SyncIns()`	Flush the pipeline and remove any unexecuted syllables from the instruction fetch logic, see*Section 12.4.3: Instruction cache control operations on page 116*.
`WaitForLink()`	Wait until the atomic sequence lock bit is cleared, or until an interrupt occurs, see *waitl on page 138*.
`Retention()`	Place the ST240 into a low power state as defined in*Section 16.3: Retention mode on page 154*.

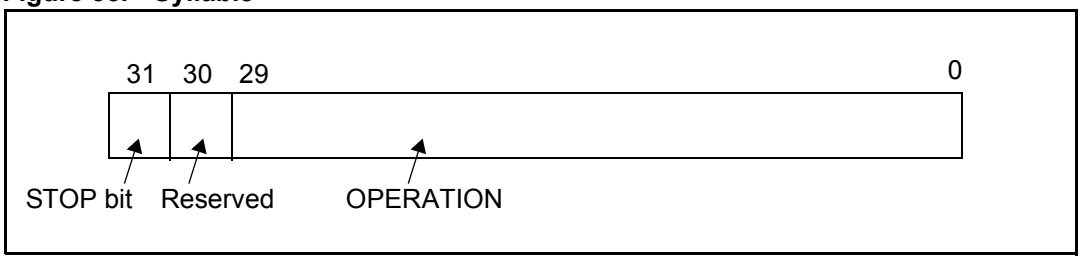
24 Instruction set

This chapter contains descriptions of all the operations and macros (pseudo-operations) in the ST240 instruction set. *Section 24.1: Bundle encoding* describes how operations are encoded in the context of bundles.

24.1 Bundle encoding

An instruction bundle consists of between one and four consecutive 32-bit words, known as syllables. Each syllable encodes either an operation or an extended immediate value. The most significant bit of each syllable (bit 31) is a **stop bit**, which is set for a syllable to indicate that it is the last syllable in the bundle, as shown in *Figure 35*.

Figure 35. Syllable



24.1.1 Extended immediates

Many operations have an **Immediate** form. In general, only small (that is, 9-bit) immediates can be directly encoded within a single word syllable. If a larger immediate is required, the instruction uses an immediate extension. This extension is encoded in an adjacent word in the bundle, making the operation effectively a two-word operation.

These immediate extensions are associated either with the operation to their left or their right in the bundle. Bit 23 is used to indicate the association:

- 0 indicates left association (word address - 1) (**imml**)
- 1 indicates right association (word address + 1) (**immr**)

The semantic descriptions of **Immediate** form operations use the following function to take into account possible immediate extensions:

Table 160. Extended immediate functions

Function	Description
`Imm(i)`	Given short immediate value **i**, return an integer value that represents the full immediate.

This function effectively performs the following:

If there is an **immr** word to the left (word address - 1) or an **imml** word to the right (word address + 1) in the bundle, then **Imm** returns:

```
(ZeroExtend23(extension) << 9) + ZeroExtend9(i);
```

Where `extension` represents the lower 23 bits of the associated **immr** or **imml**.

Otherwise **Imm** returns:

```
SignExtend9(i);
```

24.1.2 Encoding restrictions

There are a number of restrictions placed on the encoding of bundles. The assembler is responsible for ensuring that these restrictions are obeyed, see *Chapter 5: Bundling rules on page 31*.

24.2 Operation specifications

The specification of each operation contains the following fields.

- Name: the name of the operation with an optional subscript. The subscript distinguishes between operations with different operand types. For example, integer operations can have either **Register** and **Immediate** formats. If no subscript exists for an operation, then it has only one format.
- Syntax: presents the assembly syntax of the operation.
- Encoding: the binary encoding is summarized in a table. It shows which bits are used for the opcode, which bits are reserved (empty fields) and which bit fields encode the operands. The operands are either register designators or immediate constants.
- Semantics: a table containing the statements (*Section 23.3: Statements on page 209*) that define the operation. The notation used is defined in *Chapter 23: Specification notation on page 202*. The table is divided into two parts by the commit point, see *Chapter 22: Execution model on page 199*.

Pre-commit phase: ● No architectural state of the machine is updated. ● Any recoverable exceptions are thrown here.
Commit phase - executed if no exceptions have been thrown: ● All architectural state is updated. ● Any exceptions thrown here are non-recoverable.

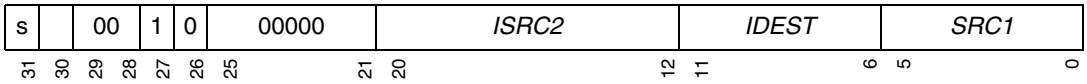
←Commit point

- Description: a brief textual description of the operation.
- Restrictions: contains any details of restrictions, which may be of the following types:
  - Address/bundle: In encoding a bundle with the operation there are a number of possible restrictions which may apply. They are detailed in *Section 24.1.2: Encoding restrictions on page 223*.
  - Latency: certain operands have latency constraints that must be observed.
  - Destination restrictions: certain operations are not allowed to use the link register (LR) as a destination.
- Exceptions: if this operation is able to throw any exceptions, they will be listed here. The semantics of the operation detail how and when they are thrown.

24.3 Example operations

add Immediate

add R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 + operand2;

R_{IDEST} ←Register(result1);

Description: Scalar 32-bit addition. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

Interpretation

The operation is given the subscript **Immediate** to indicate that one of its source operands is an immediate rather than both operands being registers.

The next line of the description shows the assembly syntax of the operation.

Just below is the binary encoding table with fields showing:

- an **s** in bit 31, which represents the **stop bit** (see *Section 24.1: Bundle encoding on page 222*)
- an unused bit (bit 30)
- the opcode, bits 29:21, which in this case is 001000000
- the operands:
  - the 9-bit immediate constant, bits 20:12
  - the destination register designator, bits 11:6
  - the source register designator, bits 5:0

The semantics table specifies the effects of the executing this operation. The table is divided into two parts. The first half contains statements which do not affect the architectural state of the machine (before the commit point). The second half contains statements that will not be executed if an exception occurs in the bundle (after the commit point).

The statements themselves are organized into three stages as follows:

1. The first two statements read the required source information:

   ```
   operand1 <- SignExtend32(RSRC1);
   operand2 <- Imm(ISRC2);
   ```

   The first statement reads the value of the R_{SRC1} register, interprets it as a signed 32-bit value and assigns this to a temporary integer called **operand1**. The second statement passes the value of ***ISRC2*** to the immediate handling function **Imm** (*Section 24.1.1: Extended immediates*). The result of the function is interpreted as a signed 32-bit value and assigned to a temporary integer called **operand2**.

2. The next statement implements the addition:

   ```
   result <- operand1 + operand2;
   ```

   This statement does not refer to any architectural state. It adds the two integers **operand1** and **operand2** together, and assigns the result to a temporary integer called **result**.

Note: Since this is a conventional mathematical addition, the result can contain more significant bits of information than the sources.

3. The final statement, executed if no exceptions have been thrown in the bundle, updates the architectural state:

   ```
   RIDEST <- Register(result);
   ```

   The function **`Register`** (see *Section 23.2.6: Single-value functions on page 207*) converts the integer **result** back to a bit field, discarding any redundant higher bits. This value is then assigned to the R_{IDEST} register.

Following the semantic description is a simple textual description of the operation.

The restrictions section shows that this operation has no restrictions. This means that up to four of these operations can be used in a bundle, and that all operands are ready for use by operations in the next bundle.

Finally, this operation cannot generate any exceptions.

24.4 Macros

Table 161 is a list of the currently implemented pseudo-operations or 'macros'. Each macro is essentially a simplified synomym for another, less intuitive operation.

Table 161. Macros

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bswap	s		01		1	0	1	0	000			000011011									*IDEST*						*SRC1*				
	perm.pb $R_{IDEST} = R_{SRC1}$, 27																															
convbi	s		01		1	1	01		*SCOND*			000000001									*IDEST*						000000				
	slctf $R_{IDEST} = B_{SCOND}$, R_0, 1																															
convib	s		00		0	1	1		1100			000000						*BDEST 2*			000000						*SRC1*				
	orl $B_{BDEST2} = R_{SRC1}$, R_0																															
$cmpge_{Cmp3R\_Reg}$	s		00		0	1	0		0110			000					*DEST*				*SRC2*						*SRC1*				
	cmple $R_{DEST} = R_{SRC1}$, R_{SRC2}																															
$cmpge_{Cmp3R\_Br}$	s		00		0	1	1		0110			000000						*BDEST 2*			*SRC2*						*SRC1*				
	cmple $B_{BDEST2} = R_{SRC1}$, R_{SRC2}																															
$cmpgeu_{Cmp3R\_Reg}$	s		00		0	1	0		0111			000					*DEST*				*SRC2*						*SRC1*				
	cmpleu $R_{DEST} = R_{SRC1}$, R_{SRC2}																															
$cmpgeu_{Cmp3R\_Br}$	s		00		0	1	1		0111			000000						*BDEST 2*			*SRC2*						*SRC1*				
	cmpleu $B_{BDEST2} = R_{SRC1}$, R_{SRC2}																															
$cmpgt_{Cmp3R\_Reg}$	s		00		0	1	0		1000			000					*DEST*				*SRC2*						*SRC1*				
	cmplt $R_{DEST} = R_{SRC1}$, R_{SRC2}																															
$cmpgt_{Cmp3R\_Br}$	s		00		0	1	1		1000			000000						*BDEST 2*			*SRC2*						*SRC1*				
	cmplt $B_{BDEST2} = R_{SRC1}$, R_{SRC2}																															
$cmpgtu_{Cmp3R\_Reg}$	s		00		0	1	0		1001			000					*DEST*				*SRC2*						*SRC1*				
	cmpltu $R_{DEST} = R_{SRC1}$, R_{SRC2}																															
$cmpgtu_{Cmp3R\_Br}$	s		00		0	1	1		1001			000000						*BDEST 2*			*SRC2*						*SRC1*				
	cmpltu $B_{BDEST2} = R_{SRC1}$, R_{SRC2}																															
$cmplef.n_{Cmp3R\_Reg}$	s		00		0	1	0		0010			000					*DEST*				*SRC2*						*SRC1*				
	cmpgef.n $R_{DEST} = R_{SRC1}$, R_{SRC2}																															
$cmplef.n_{Cmp3R\_Br}$	s		00		0	1	1		0010			000000						*BDEST 2*			*SRC2*						*SRC1*				
	cmpgef.n $B_{BDEST2} = R_{SRC1}$, R_{SRC2}																															
$cmpltf.n_{Cmp3R\_Reg}$	s		00		0	1	0		0100			000					*DEST*				*SRC2*						*SRC1*				
	cmpgtf.n $R_{DEST} = R_{SRC1}$, R_{SRC2}																															
$cmpltf.n_{Cmp3R\_Br}$	s		00		0	1	1		0100			000000						*BDEST 2*			*SRC2*						*SRC1*				
	cmpgtf.n $B_{BDEST2} = R_{SRC1}$, R_{SRC2}																															
idle	1		11		0	001			0			00000000000000000000																				
	goto 0																															

Table 161. Macros (Continued)

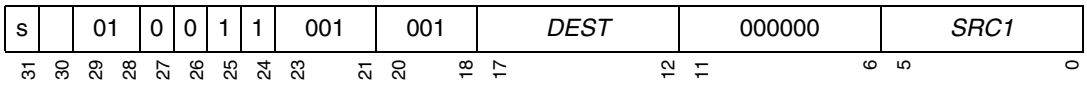
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
mfb	s		01		1	1	01		*SCOND*			000000001									*IDEST*						000000					
	slctf $R_{IDEST} = B_{SCOND}$, R_0, 1																															
mov_{Int3R}	s		00		0	0	00000					000			*DEST*						*SRC2*						000000					
	add $R_{DEST} = R_0$, R_{SRC2}																															
mov_{Int3I}	s		00		1	0	00000					*ISRC2*									*IDEST*						000000					
	add $R_{IDEST} = R_0$, ISRC2																															
mtb	s		00		0	1	1	1100				000000						*BDEST2*			000000						*SRC1*					
	orl $B_{BDEST2} = R_{SRC1}$, R_0																															
mull	s		00		1	0	10110					*ISRC2*									*IDEST*						*SRC1*					
	mul32 $R_{NLIDEST} = R_{SRC1}$, ISRC2																															
nop	s		00		0	0	00000					000			000000						000000						000000					
	add $R_0 = R_0$, R_0																															
slctf	s		01		0	1	10		*SCOND*			001			*DEST*						*SRC2*						*SRC1*					
	slct $R_{DEST} = B_{SCOND}$, R_{SRC1}, R_{SRC2}																															
sxtb	s		01		1	0	0	0	101			000001000									*IDEST*						*SRC1*					
	sxt $R_{IDEST} = R_{SRC1}$, 8																															
sxth	s		01		1	0	0	0	101			000010000									*IDEST*						*SRC1*					
	sxt $R_{IDEST} = R_{SRC1}$, 16																															
zxtb	s		00		1	0	01001					011111111									*IDEST*						*SRC1*					
	and $R_{IDEST} = R_{SRC1}$, 255																															
zxth	s		01		1	0	0	0	110			000010000									*IDEST*						*SRC1*					
	zxt $R_{IDEST} = R_{SRC1}$, 16																															
slctf.pb	s		01		0	1	10		*SCOND*			000			*DEST*						*SRC2*						*SRC1*					
	slct.pb $R_{DEST} = B_{SCOND}$, R_{SRC1}, R_{SRC2}																															
unpacku.pbh	s		01		0	0	1	0	000			001			*DEST*						000000						*SRC1*					
	shuff.pbh $R_{DEST} = R_{SRC1}$, R_0																															
unpacku.pbl	s		01		0	0	1	0	001			001			*DEST*						000000						*SRC1*					
	shuff.pbl $R_{DEST} = R_{SRC1}$, R_0																															
pack.ph	s		01		0	0	1	1	001			010			*DEST*						*SRC2*						*SRC1*					
	shuff.phl $R_{DEST} = R_{SRC1}$, R_{SRC2}																															

24.5 Operations

This section specifies all the operations in the instruction set. For ease of use, the operations are listed in alphabetical order. The semantics of each operation is written using the notational language defined in *Chapter 23: Specification notation on page 202*.

abss.ph Register

abss.ph R_{DEST} = R_{SRC1}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd←SignedExtract16(operand1, j);
        IF (unpacked_opd< 0)
                unpacked_opd←(-unpacked_opd);
        unpacked_opd←Saturate16(unpacked_opd);
        result1 ←result1 ∨  MaskAndShift16(unpacked_opd, j);
}
```

```
RDEST ←Register(result1);
```

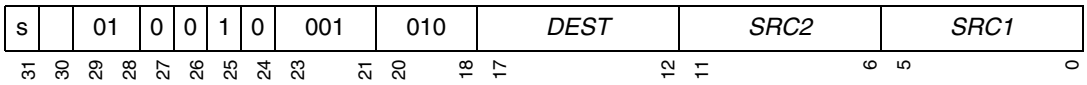
Description: Packed 16-bit signed absolute with saturation. Operands may be signed 16-bit integers or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

absubu.pb Register

absubu.pb R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        subresult ←UnsignedExtract8(operand1, j) - UnsignedExtract8(operand2, j);
        IF (subresult > 0)
                result1 ←result1 ∨ MaskAndShift8(subresult, j);
        ELSE
                result1 ←result1 ∨ MaskAndShift8(-subresult, j);
}
```

```
RDEST ←Register(result1);
```

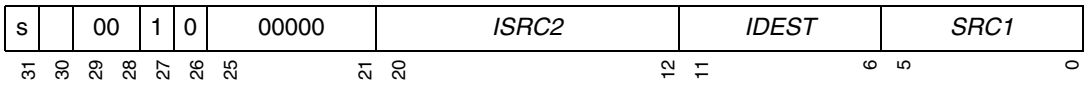
Description: Packed 8-bit unsigned absolute difference.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

add Immediate

add R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 + operand2;

R_{IDEST} ←Register(result1);

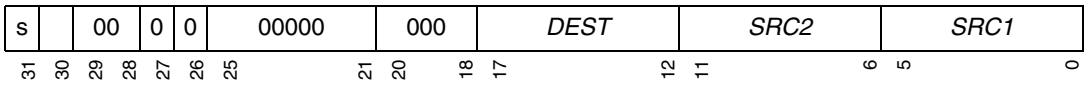
Description: Scalar 32-bit addition. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

add Register

add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 + operand2;

R_{DEST} ←Register(result1);

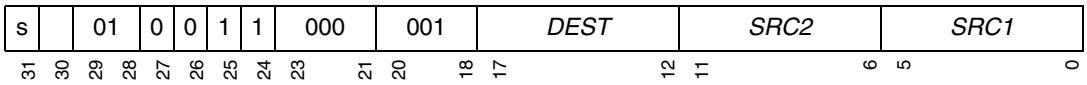
Description: Scalar 32-bit addition. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

add.ph Register

add.ph R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        addresult ←SignedExtract16(operand1, j) + SignedExtract16(operand2, j);
        result1 ←result1 ∨  MaskAndShift16(addresult, j);
}
```

```
RDEST ←Register(result1);
```

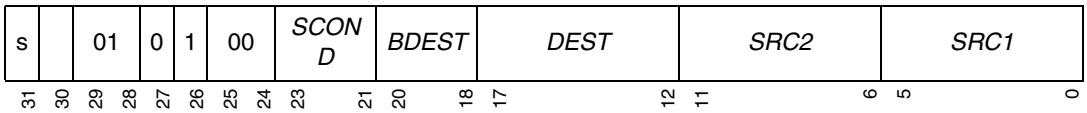
Description: Packed 16-bit addition. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

addcg

addcg R_{DEST}, B_{BDEST} = R_{SRC1}, R_{SRC2}, B_{SCOND}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
operand3 ←ZeroExtend$_{4}$(B_{SCOND});
result1 ←(operand1 + operand2) + (operand3 ≠ 0);
result2 ←Bit(result1, 32);

R_{DEST} ←Register(result1);
B_{BDEST} ←BranchRegister(result2);

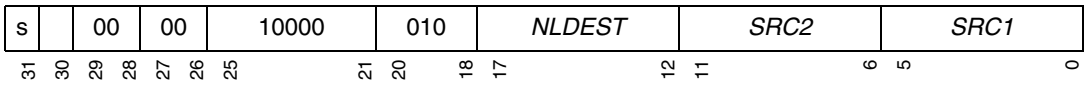
Description: 32bit scalar addition with carry input and generate a carry output. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST} is available for reading by br/brf.
No other latency restrictions before B_{BDEST} is available for reading by any other operation.

Exceptions: None.

addf.n Floating point - Register

addf.n R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});

result1 ←FAddSNonIeee(operand1, operand2);
R_{NLDEST} ←Register(result1);

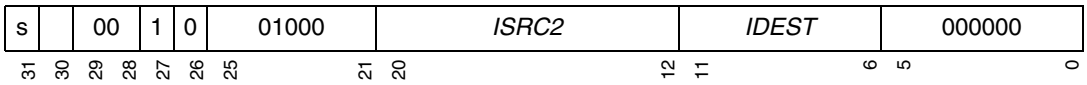
Description: IEEE754 format single precision floating point addition.

Restrictions: Must be encoded at an even word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

addpc Immediate

addpc R_{IDEST} **=** ISRC2



Semantics:

operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←ZeroExtend$_{32}$(BUNDLE_PC) + operand2;

R_{IDEST} ←Register(result1);

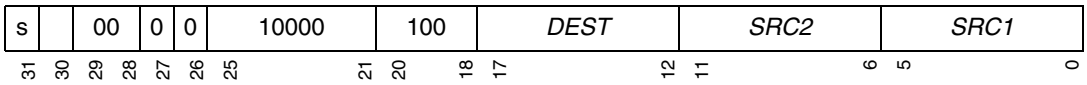
Description: Scalar 32bit addition of immediate value and virtual PC of the current bundle.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

adds Register

adds R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←Saturate$_{32}$(operand1 + operand2);

R_{DEST} ←Register(result1);

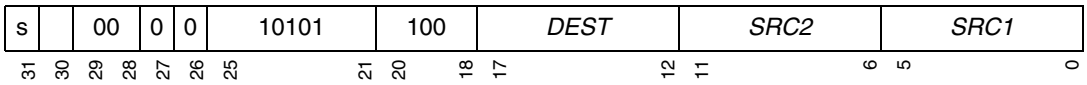
Description: Scalar 32-bit addition with saturation. Operands may be signed or unsigned integers or fractional 1.31 values.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

adds.ph Register

adds.ph R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        addresult ←SignedExtract16(operand1, j) + SignedExtract16(operand2, j);
        addresult ←Saturate16(addresult);
        result1 ←result1 ∨  MaskAndShift16(addresult, j);
}
```

```
RDEST ←Register(result1);
```

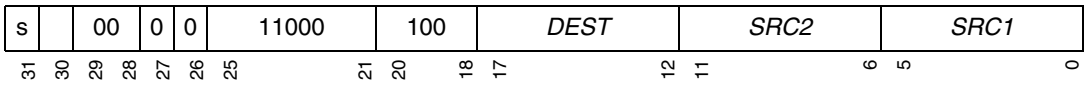
Description: Packed 16-bit signed absolute with saturation. Operands may be signed 16-bit integers or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

addso Register

addso R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←Overflow$_{32}$(operand1 + operand2);

R_{DEST} ←Register(result1);

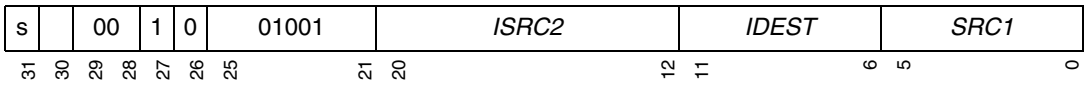
Description: Indicates whether an adds operation with the given input operands causes a saturation.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

and Immediate

and R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ∧ operand2;

R_{IDEST} ←Register(result1);

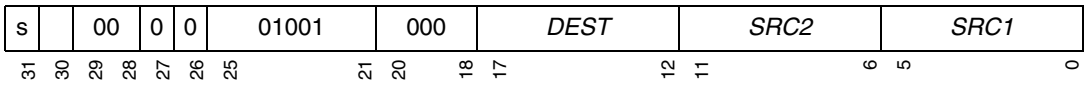
Description: Bitwise AND.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

and Register

and R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 ∧ operand2;

R_{DEST} ←Register(result1);

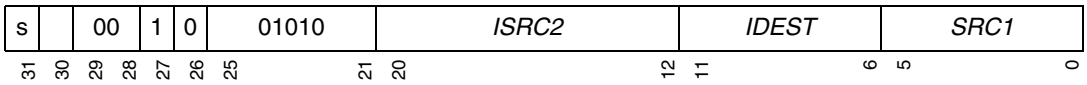
Description: Bitwise AND.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

andc Immediate

andc R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←(~operand1) ∧ operand2;

R_{IDEST} ←Register(result1);

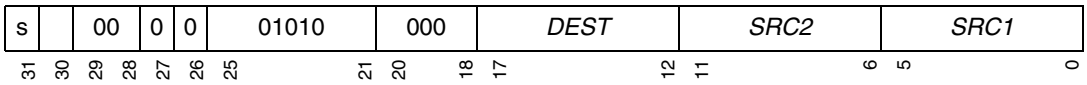
Description: Negate operand 1 and then bitwise AND.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

andc Register

andc R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(~operand1) ∧ operand2;

R_{DEST} ←Register(result1);

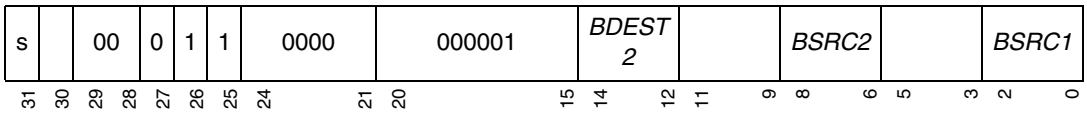
Description: Negate operand 1 and then bitwise AND.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

andl Branch Register - Branch Register

andl B_{BDEST2} = B_{BSRC1}, B_{BSRC2}



Semantics:

operand1 ←ZeroExtend$_4$(B_{BSRC1});
operand2 ←ZeroExtend$_4$(B_{BSRC2});
result1 ←(operand1 ≠ 0) AND (operand2 ≠ 0);

B_{BDEST2} ←BranchRegister(result1);

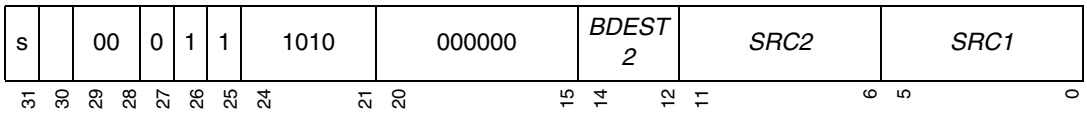
Description: Logical AND.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

andl Branch Register - Register

andl B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 ≠ 0) AND (operand2 ≠ 0);

B_{BDEST2} ←BranchRegister(result1);

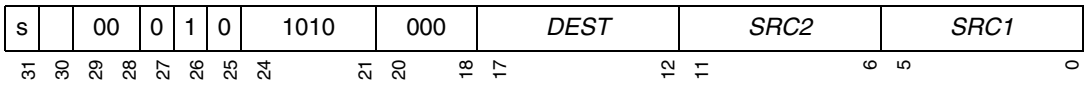
Description: Logical AND.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

andl Register - Register

andl R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 ≠ 0) AND (operand2 ≠ 0);

R_{DEST} ←Register(result1);

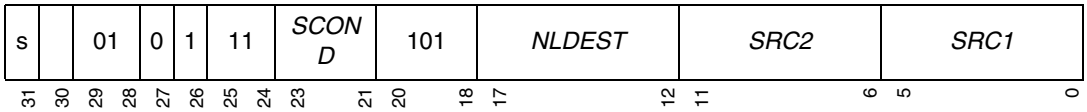
Description: Logical AND.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

avg4u.pb Register

avg4u.pb R_{NLDEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend4(BSCOND);
operand2 ←ZeroExtend32(RSRC1);
operand3 ←ZeroExtend32(RSRC2);
sum1 ←ZeroExtend2(operand1);
sum2 ←sum1;
REPEAT j FROM 0 FOR 2 {
        sum1 ←sum1 + UnsignedExtract8(operand2, j);
        sum1 ←sum1 + UnsignedExtract8(operand3, j);
        sum2 ←sum2 + UnsignedExtract8(operand2, j + 2);
        sum2 ←sum2 + UnsignedExtract8(operand3, j + 2);
}
result1 ←(sum1 >> 2) ∨  MaskAndShift16( sum2 >> 2, 1);
```

```
RNLDEST ←Register(result1);
```

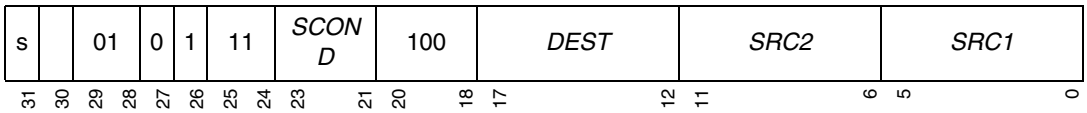
Description: Packed 8-bit unsigned 4-way average with selectable rounding mode (round zero, round nearest negative, round nearest positive or round positive).

Restrictions: No address or bundle restrictions.
R_{NLDEST} can be any general register except the link register.
There is a latency of 1 cycle before R_{NLDEST} is available for reading.

Exceptions: None.

avgu.pb Register

avgu.pb R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend4(BSCOND);
operand2 ←ZeroExtend32(RSRC1);
operand3 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        sum ←( operand1 ≠ 0 );
        sum ←sum + UnsignedExtract8(operand2, j);
        sum ←sum + UnsignedExtract8(operand3, j);
        result1 ←result1 ∨  MaskAndShift8(sum >> 1, j);
}
```

```
RDEST ←Register(result1);
```

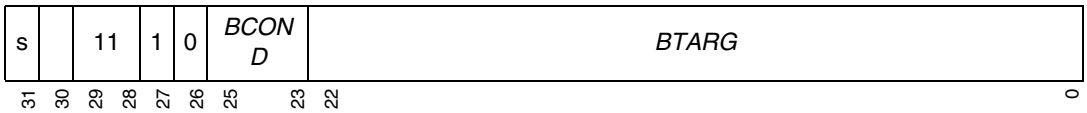
Description: Packed 8-bit unsigned average with selectable rounding mode (round zero or round positive).

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

br

br B_{BCOND}, BTARG



Semantics:

```
operand1 ←ZeroExtend4(BBCOND);
operand2 ←SignExtend23(BTARG)<< 2;
IF (operand1 ≠ 0)
        PC ←Register(ZeroExtend32(BUNDLE_PC) + operand2);
```

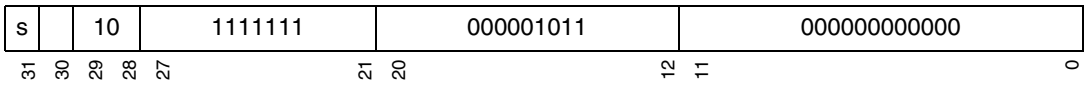
Description: Branch.

Restrictions: Must be the first syllable of a bundle.
No latency constraints.

Exceptions: None.

break

break



Semantics:

```
THROW ILL_INST;
```

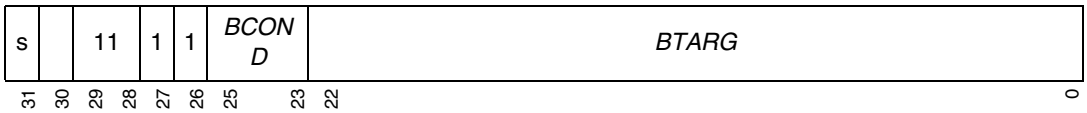
Description: Causes illegal instruction exception.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: ILL_INST

brf

brf B_{BCOND}, BTARG



Semantics:

```
operand1 ←ZeroExtend4(BBCOND);
operand2 ←SignExtend23(BTARG)<< 2;
IF (operand1 = 0)
        PC ←Register(ZeroExtend32(BUNDLE_PC) + operand2);
```

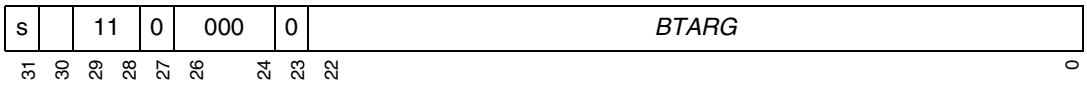
Description: Branch false.

Restrictions: Must be the first syllable of a bundle.
No latency constraints.

Exceptions: None.

call Immediate

call $r63 = BTARG



Semantics:

operand1 ←SignExtend$_{23}$(BTARG)<< 2;
NEXT_PC←PC;
PC ←Register(ZeroExtend$_{32}$(BUNDLE_PC) + operand1);
LR ←NEXT_PC;

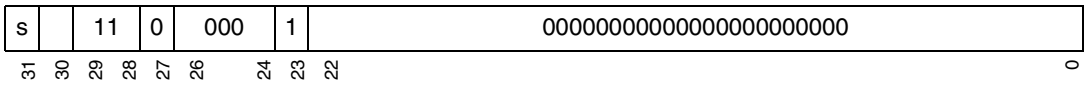
Description: Jump and link.

Restrictions: Must be the first syllable of a bundle.
No latency constraints.

Exceptions: None.

call Link Register

call $r63 = $r63



Semantics:

NEXT_PC←PC;
PC ←Register(ZeroExtend$_{32}$(LR));
LR ←NEXT_PC;

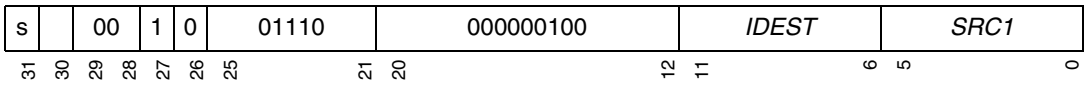
Description: Jump (using Link Register) and link.

Restrictions: Must be the first syllable of a bundle.
There are no latency constraints between a call updating the LR and this operation.
There is a latency of 3 cycles between a load writing to the LR and this operation.
There is a latency of 2 cycles between any other operation updating the LR and this operation.

Exceptions: None.

clz

clz R_{IDEST} = R_{SRC1}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
result1 ←CountLeadingZeros(operand1);

R_{IDEST} ←Register(result1);

Description: Count leading zeros.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpeq Branch Register - Immediate

cmpeq B_{IBDEST} **=** R_{SRC1}**,** ISRC2

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 = operand2;

B_{IBDEST} ←BranchRegister(result1);

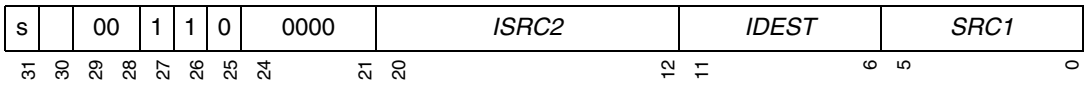
Description: Test for equality.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{IBDEST} is available for reading by br/brf.
No other latency restrictions before B_{IBDEST} is available for reading by any other operation.

Exceptions: None.

cmpeq Register - Immediate

cmpeq R_{IDEST} **=** R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 = operand2;

R_{IDEST} ←Register(result1);

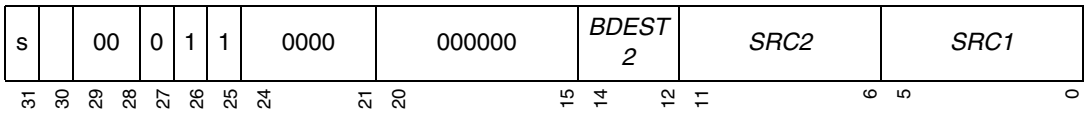
Description: Test for equality.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpeq Branch Register - Register

cmpeq B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 = operand2;

B_{BDEST2} ←BranchRegister(result1);

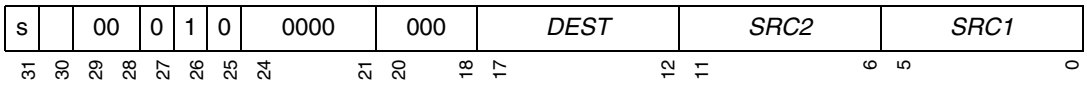
Description: Test for equality.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmpeq Register - Register

cmpeq R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 = operand2;

R_{DEST} ←Register(result1);

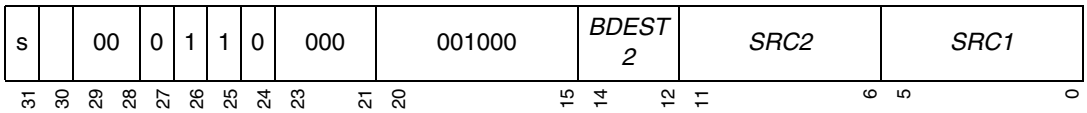
Description: Test for equality.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpeq.pb Branch Register - Register

cmpeq.pb B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        unpacked_opd1←UnsignedExtract8(operand1, j);
        unpacked_opd2←UnsignedExtract8(operand2, j);
        unpacked_res←(unpacked_opd1= unpacked_opd2);
        result1 ←result1 ∨  MaskAndShift1(unpacked_res, j);
}
```

B_{BDEST2} ←BranchRegister(result1);

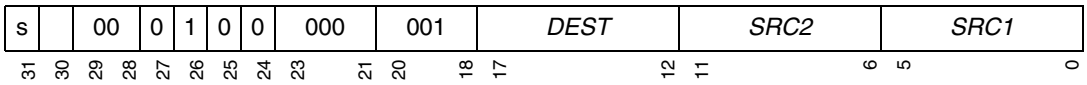
Description: Packed 8-bit test for equality writing result to branch register.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmpeq.pb Register

cmpeq.pb R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        unpacked_opd1←SignedExtract8(operand1, j);
        unpacked_opd2←SignedExtract8(operand2, j);
        unpacked_res←(unpacked_opd1= unpacked_opd2);
        result1 ←result1 ∨ MaskAndShift8(unpacked_res, j);
}
```

```
RDEST ←Register(result1);
```

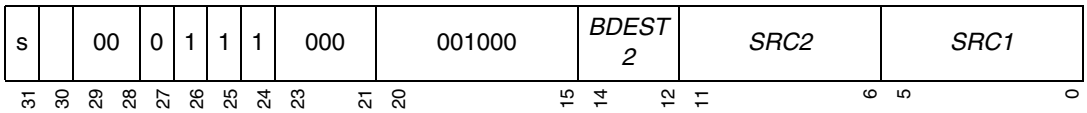
Description: Packed 8-bit test for equality.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpeq.ph Branch Register - Register

cmpeq.ph B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←SignedExtract16(operand1, j);
        unpacked_opd2←SignedExtract16(operand2, j);
        unpacked_res←SignExtend1(unpacked_opd1= unpacked_opd2);
        result1 ←result1 ∨  MaskAndShift2(unpacked_res, j);
}
```

B_{BDEST2} ←BranchRegister(result1);

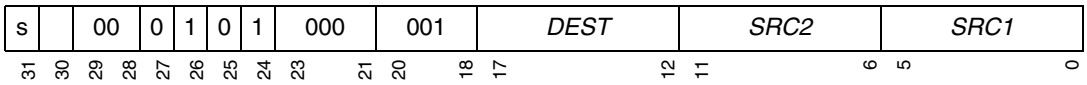
Description: Packed 16-bit test for equality to branch register.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmpeq.ph Register

cmpeq.ph R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←SignedExtract16(operand1, j);
        unpacked_opd2←SignedExtract16(operand2, j);
        unpacked_res←(unpacked_opd1= unpacked_opd2);
        result1 ←result1 ∨  MaskAndShift16(unpacked_res, j);
}
```

```
RDEST ←Register(result1);
```

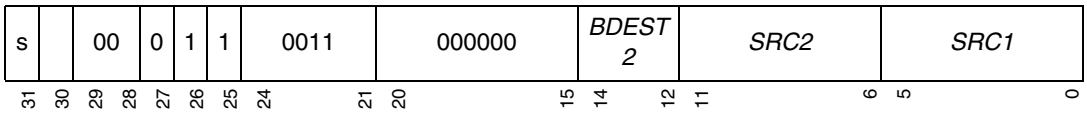
Description: Packed 16-bit test for equality.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpeqf.n Branch Register - Register

cmpeqf.n B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
result1 ←FCompEqNonIeee(operand1, operand2);

B_{BDEST2} ←BranchRegister(result1);

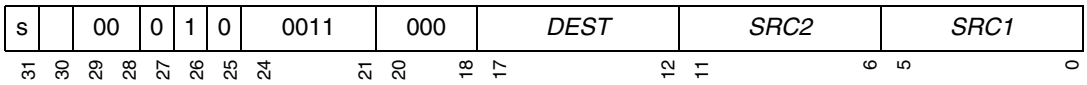
Description: IEEE754 format single precision floating point equality comparison.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmpeqf.n Register - Register

cmpeqf.n R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
result1 ←FCompEqNonIeee(operand1, operand2);

R_{DEST} ←Register(result1);

Description: IEEE754 format single precision floating point equality comparison.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpge Branch Register - Immediate

cmpge B_{IBDEST} = R_{SRC1}, ISRC2

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ≥ operand2;

B_{IBDEST} ←BranchRegister(result1);

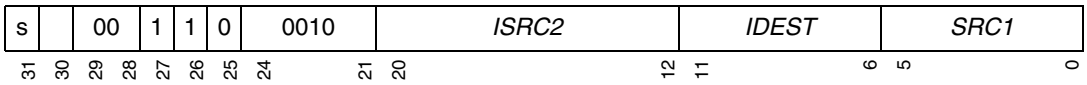
Description: Signed compare equal or greater than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{IBDEST} is available for reading by br/brf.
No other latency restrictions before B_{IBDEST} is available for reading by any other operation.

Exceptions: None.

cmpge Register - Immediate

cmpge R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ≥ operand2;

R_{IDEST} ←Register(result1);

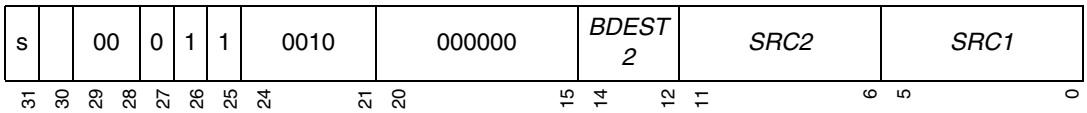
Description: Signed compare equal or greater than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpgef.n Branch Register - Register

cmpgef.n B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←FCompGENonIeee(operand1, operand2);

B_{BDEST2} ←BranchRegister(result1);

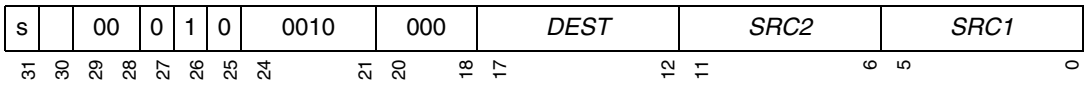
Description: Floating point compare equal or greater than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmpgef.n Register - Register

cmpgef.n R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←FCompGENonIeee(operand1, operand2);

R_{DEST} ←Register(result1);

Description: Floating point compare equal or greater than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpgeu Branch Register - Immediate

cmpgeu B_{IBDEST} = R_{SRC1}, ISRC2

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ≥ operand2;

B_{IBDEST} ←BranchRegister(result1);

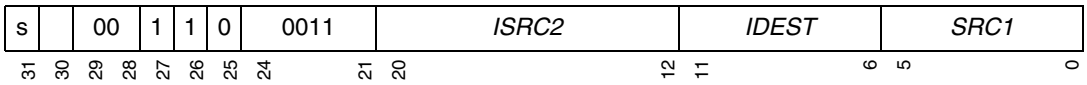
Description: Unsigned compare equal or greater than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{IBDEST} is available for reading by br/brf.
No other latency restrictions before B_{IBDEST} is available for reading by any other operation.

Exceptions: None.

cmpgeu Register - Immediate

cmpgeu R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ≥ operand2;

R_{IDEST} ←Register(result1);

Description: Unsigned compare equal or greater than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpgt Branch Register - Immediate

cmpgt B_{IBDEST} = R_{SRC1}, ISRC2

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 > operand2;

B_{IBDEST} ←BranchRegister(result1);

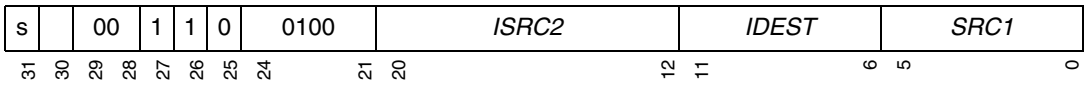
Description: Signed compare greater than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{IBDEST} is available for reading by br/brf.
No other latency restrictions before B_{IBDEST} is available for reading by any other operation.

Exceptions: None.

cmpgt Register - Immediate

cmpgt R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 > operand2;

R_{IDEST} ←Register(result1);

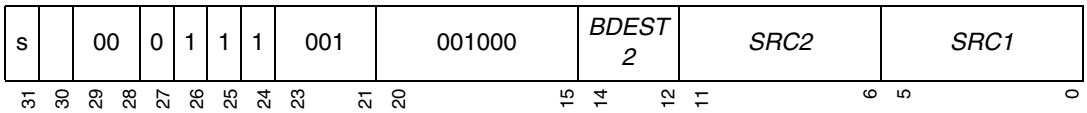
Description: Signed compare greater than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpgt.ph Branch Register - Register

cmpgt.ph B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←SignedExtract16(operand1, j);
        unpacked_opd2←SignedExtract16(operand2, j);
        unpacked_res←SignExtend1(unpacked_opd1> unpacked_opd2);
        result1 ←result1 ∨  MaskAndShift2(unpacked_res, j);
}
```

B_{BDEST2} ←BranchRegister(result1);

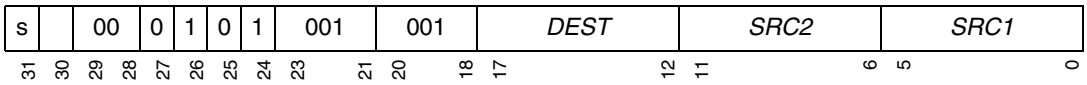
Description: Packed 16-bit greater than test to branch register.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmpgt.ph Register

cmpgt.ph R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←SignedExtract16(operand1, j);
        unpacked_opd2←SignedExtract16(operand2, j);
        unpacked_res←(unpacked_opd1> unpacked_opd2);
        result1 ←result1 ∨  MaskAndShift16(unpacked_res, j);
}
```

```
RDEST ←Register(result1);
```

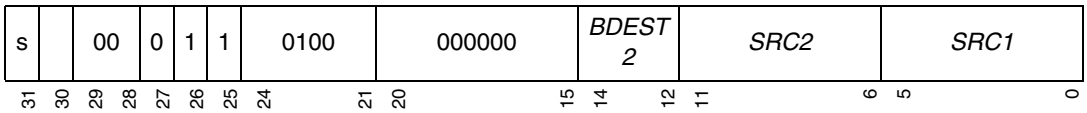
Description: Packed 16-bit signed greater than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpgtf.n Branch Register - Register

cmpgtf.n B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
result1 ←FCompGtNonIeee(operand1, operand2);
```

```
BBDEST2 ←BranchRegister(result1);
```

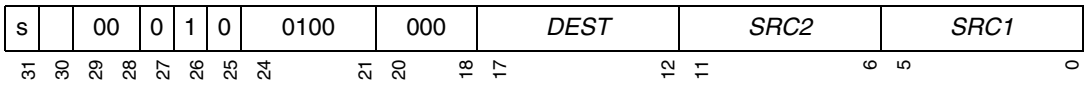
Description: IEEE754 format single precision floating point greater than comparison.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmpgtf.n Register - Register

cmpgtf.n R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←FCompGtNonIeee(operand1, operand2);

R_{DEST} ←Register(result1);

Description: IEEE754 format single precision floating point greater than comparison.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpgtu Branch Register - Immediate

cmpgtu B_{IBDEST} = R_{SRC1}, ISRC2

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 > operand2;

B_{IBDEST} ←BranchRegister(result1);

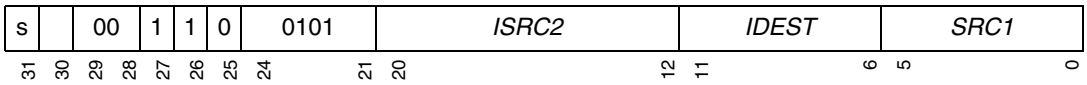
Description: Unsigned compare greater than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{IBDEST} is available for reading by br/brf.
No other latency restrictions before B_{IBDEST} is available for reading by any other operation.

Exceptions: None.

cmpgtu Register - Immediate

cmpgtu R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 > operand2;

R_{IDEST} ←Register(result1);

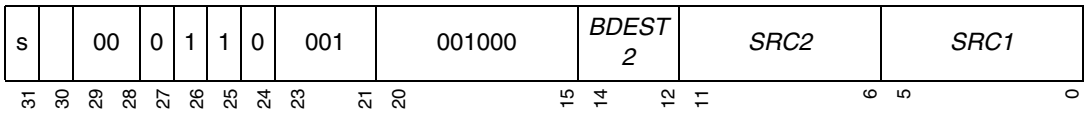
Description: Unsigned compare greater than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpgtu.pb Branch Register - Register

cmpgtu.pb B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        unpacked_opd1←UnsignedExtract8(operand1, j);
        unpacked_opd2←UnsignedExtract8(operand2, j);
        unpacked_res←(unpacked_opd1> unpacked_opd2);
        result1 ←result1 ∨  MaskAndShift1(unpacked_res, j);
}
```

B_{BDEST2} ←BranchRegister(result1);

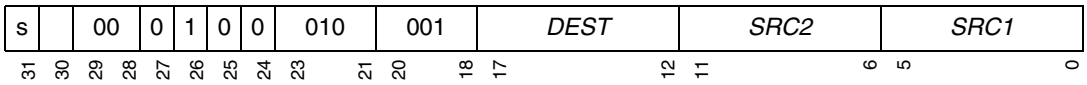
Description: Packed 8-bit unsigned greater than writing result to branch register.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmpgtu.pb Register

cmpgtu.pb R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        unpacked_opd1←UnsignedExtract8(operand1, j);
        unpacked_opd2←UnsignedExtract8(operand2, j);
        unpacked_res←(unpacked_opd1> unpacked_opd2);
        result1 ←result1 ∨  MaskAndShift8(unpacked_res, j);
}
```

```
RDEST ←Register(result1);
```

Description: Packed 8-bit unsigned greater than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmple Branch Register - Immediate

cmple B_{IBDEST} = R_{SRC1}, ISRC2

Semantics:

operand1 ←$SignExtend_{32}$(R_{SRC1});
operand2 ←$SignExtend_{32}$(Imm(ISRC2));
result1 ←operand1 ≤operand2;

B_{IBDEST} ←BranchRegister(result1);

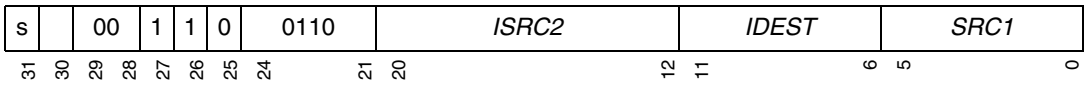
Description: Signed compare equal or less than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{IBDEST} is available for reading by br/brf.
No other latency restrictions before B_{IBDEST} is available for reading by any other operation.

Exceptions: None.

cmple Register - Immediate

cmple R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ≤operand2;

R_{IDEST} ←Register(result1);

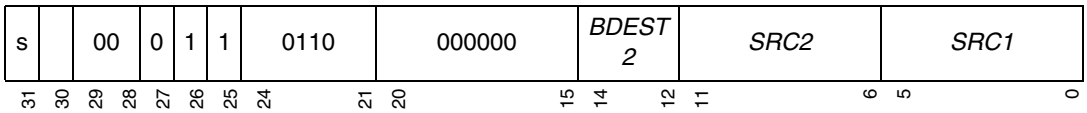
Description: Signed compare equal or less than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmple Branch Register - Register

cmple B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 ≤operand2;

B_{BDEST2} ←BranchRegister(result1);

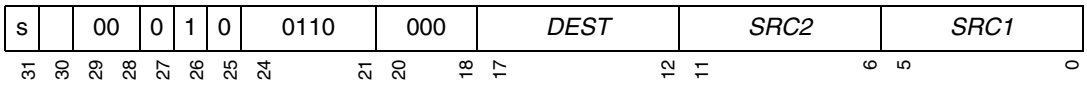
Description: Signed compare equal or less than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmple Register - Register

cmple R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 ≤operand2;

R_{DEST} ←Register(result1);

Description: Signed compare equal or less than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpleu Branch Register - Immediate

cmpleu B_{IBDEST} = R_{SRC1}, ISRC2

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ≤operand2;

B_{IBDEST} ←BranchRegister(result1);

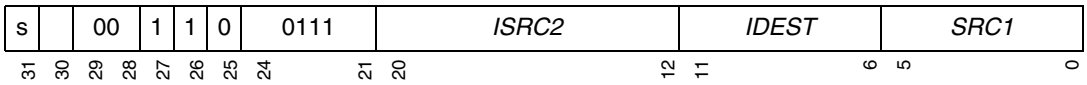
Description: Unsigned compare equal or less than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{IBDEST} is available for reading by br/brf.
No other latency restrictions before B_{IBDEST} is available for reading by any other operation.

Exceptions: None.

cmpleu Register - Immediate

cmpleu R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ≤operand2;

R_{IDEST} ←Register(result1);

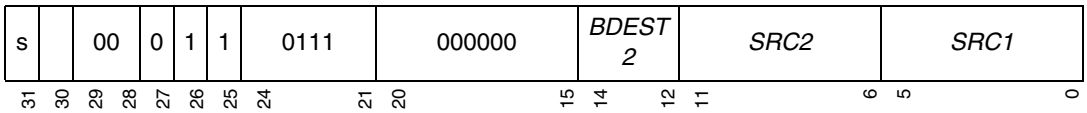
Description: Unsigned compare equal or less than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpleu Branch Register - Register

cmpleu B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
result1 ←operand1 ≤operand2;

B_{BDEST2} ←BranchRegister(result1);

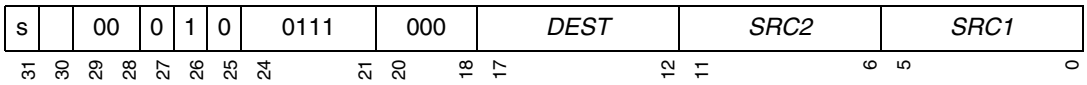
Description: Unsigned compare equal or less than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmpleu Register - Register

cmpleu R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
result1 ←operand1 ≤operand2;

R_{DEST} ←Register(result1);

Description: Unsigned compare equal or less than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmplt Branch Register - Immediate

cmplt B_{IBDEST} = R_{SRC1}, ISRC2

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 < operand2;

B_{IBDEST} ←BranchRegister(result1);

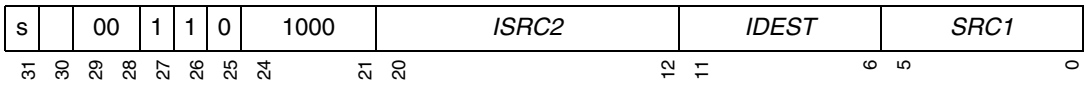
Description: Signed compare less than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{IBDEST} is available for reading by br/brf.
No other latency restrictions before B_{IBDEST} is available for reading by any other operation.

Exceptions: None.

cmplt Register - Immediate

cmplt R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 < operand2;

R_{IDEST} ←Register(result1);

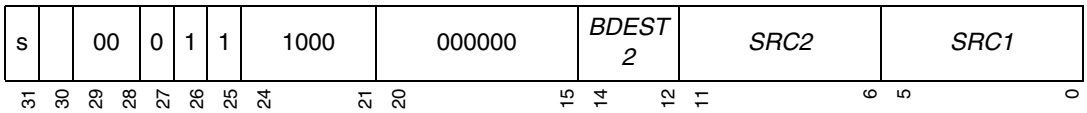
Description: Signed compare less than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmplt Branch Register - Register

cmplt B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 < operand2;

B_{BDEST2} ←BranchRegister(result1);

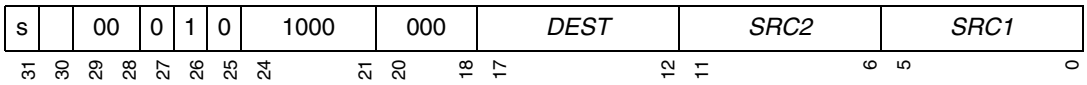
Description: Signed compare less than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmplt Register - Register

cmplt R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 < operand2;

R_{DEST} ←Register(result1);

Description: Signed compare less than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpltu Branch Register - Immediate

cmpltu B_{IBDEST} = R_{SRC1}, ISRC2

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 < operand2;

B_{IBDEST} ←BranchRegister(result1);

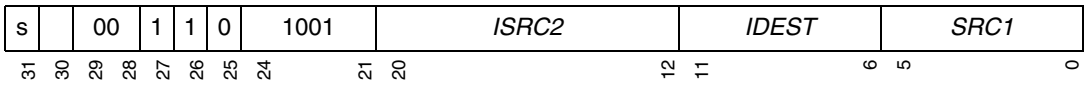
Description: Unsigned compare less than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{IBDEST} is available for reading by br/brf.
No other latency restrictions before B_{IBDEST} is available for reading by any other operation.

Exceptions: None.

cmpltu Register - Immediate

cmpltu R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 < operand2;

R_{IDEST} ←Register(result1);

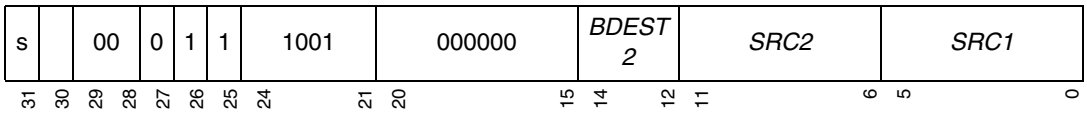
Description: Unsigned compare less than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpltu Branch Register - Register

cmpltu B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←operand1 < operand2;
```

```
BBDEST2 ←BranchRegister(result1);
```

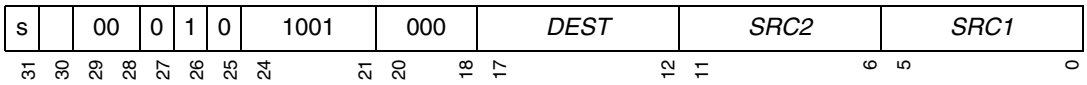
Description: Unsigned compare less than.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmpltu Register - Register

cmpltu R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
result1 ←operand1 < operand2;

R_{DEST} ←Register(result1);

Description: Unsigned compare less than.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpne Branch Register - Immediate

cmpne B_{IBDEST} **=** R_{SRC1}**,** ISRC2

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ≠ operand2;

B_{IBDEST} ←BranchRegister(result1);

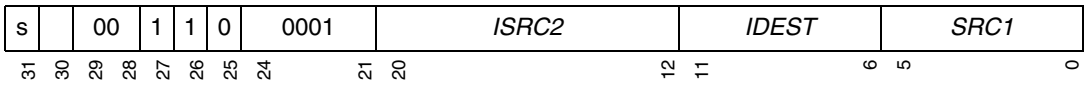
Description: Test for inequality.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{IBDEST} is available for reading by br/brf.
No other latency restrictions before B_{IBDEST} is available for reading by any other operation.

Exceptions: None.

cmpne Register - Immediate

cmpne R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ≠ operand2;

R_{IDEST} ←Register(result1);

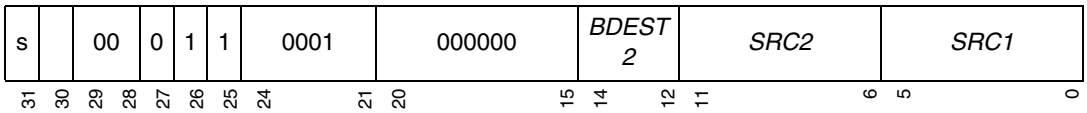
Description: Test for inequality.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

cmpne Branch Register - Register

cmpne B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 ≠ operand2;

B_{BDEST2} ←BranchRegister(result1);

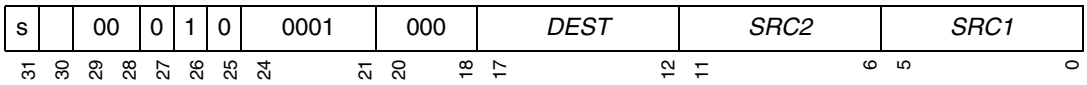
Description: Test for inequality.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

cmpne Register - Register

cmpne R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 ≠ operand2;

R_{DEST} ←Register(result1);

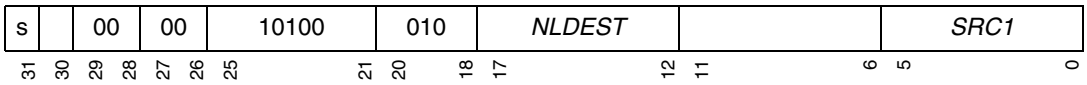
Description: Test for inequality.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

convfi.n Floating point - Register

convfi.n R_{NLDEST} = R_{SRC1}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
result1 ←SFloatToIntNonIeee(operand1);
RNLDEST ←Register(result1);
```

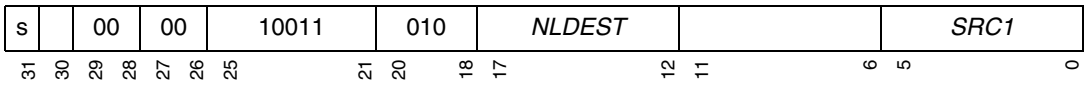
Description: IEEE754 format single precision floating point to signed integer conversion.

Restrictions: Must be encoded at an even word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

convif.n Floating point - Register

convif.n R_{NLDEST} = R_{SRC1}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
result1 ←IntToSFloatNonIeee(operand1);
RNLDEST ←Register(result1);
```

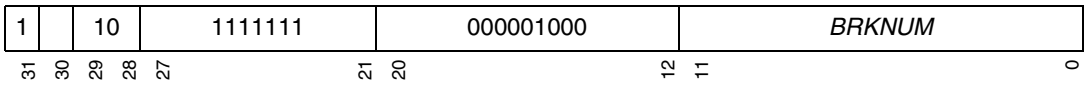
Description: Signed integer to IEEE754 format single precision floating point conversion.

Restrictions: Must be encoded at an even word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

dbgsbrk

dbgsbrk BRKNUM



Semantics:

```
operand1 ←ZeroExtend12(BRKNUM);
IF (((DBG_SBREAK_CONTROL∧ 1)≠ 0) AND (NOT(PSW[DEBUG_MODE])))
        DbgThrow(DBG_SBREAK);
ELSE
        THROW ILL_INST;
```

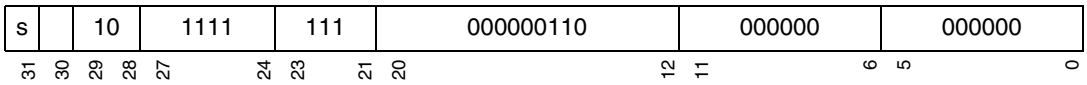
Description: Debug software breakpoint. Causes immediate entry into debug mode.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles between an operation writing DBG_SBREAK_CONTROL and this operation being issued.
There is a latency of 2 cycles between an operation writing PSW and this operation being issued.

Exceptions: ILL_INST

dib

dib



Semantics:

Dib();

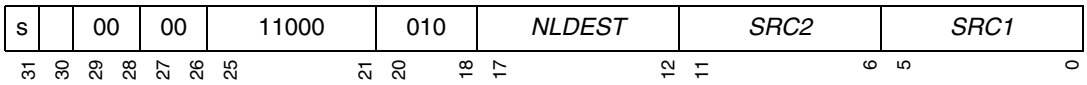
Description: Data-instruction barrier.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: None.

div Register

div R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});

result1 ←IDivIeee(operand1, operand2);
R_{NLDEST} ←Register(result1);

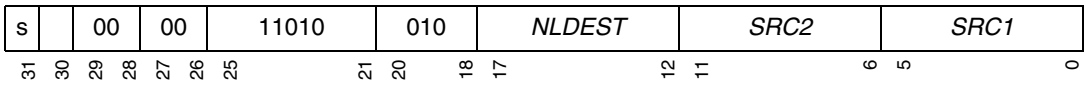
Description: Signed integer division. May stall the pipeline, see definition of IDivIeee().

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

divu Register

divu R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});

result1 ←UIDivIeee(operand1, operand2);
R_{NLDEST} ←Register(result1);

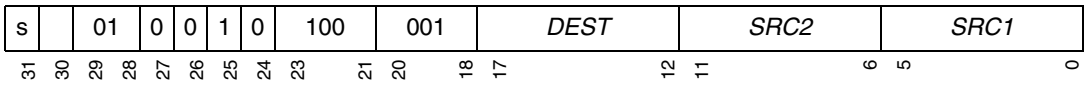
Description: Unsigned integer division. May stall the pipeline, see definition of UIDivIeee().

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

ext1.pb Register

ext1.pb R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←MaskAndShift8(operand2, 3);
REPEAT j FROM 0 FOR 3 {
        unpacked_opd1←UnsignedExtract8(operand1, j + 1);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd1, j);
}
```

```
RDEST ←Register(result1);
```

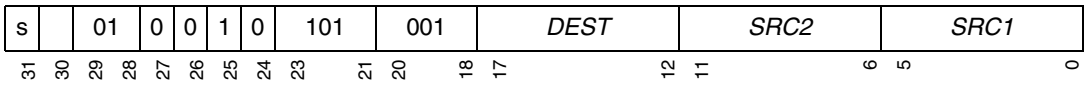
Description: Extract word starting from byte 1.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

ext2.pb Register

ext2.pb R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
result1 ←UnsignedExtract$_{16}$(operand1, 1) ∨ MaskAndShift$_{16}$(operand2, 1);

R_{DEST} ←Register(result1);

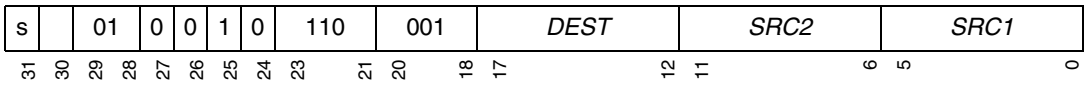
Description: Extract word starting from byte 2.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

ext3.pb Register

ext3.pb R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←UnsignedExtract8(operand1, 3);
REPEAT j FROM 0 FOR 3 {
        unpacked_opd2←UnsignedExtract8(operand2, j);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd2, j + 1);
}
```

```
RDEST ←Register(result1);
```

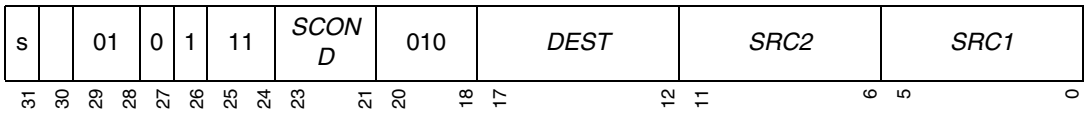
Description: Extract word starting from byte 3.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

extl.pb Register

extl.pb R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend4(BSCOND);
operand2 ←ZeroExtend32(RSRC1);
operand3 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        index ←j + ZeroExtend2(operand1);
        IF (index < 4)
                byte ←UnsignedExtract8(operand2, index);
        ELSE
                byte ←UnsignedExtract8(operand3, index - 4);
        result1 ←result1 ∨  MaskAndShift8( byte, j);
}
```

```
RDEST ←Register(result1);
```

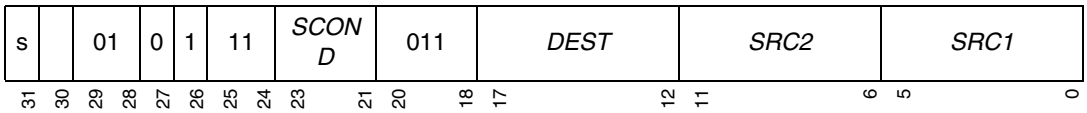
Description: Dynamic extract left operation.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

extr.pb Register

extr.pb R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend4(BSCOND);
operand2 ←ZeroExtend32(RSRC1);
operand3 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        index ←(j + 4) - ZeroExtend2(operand1);
        IF (index ≥ 4)
                byte ←UnsignedExtract8(operand2, index - 4);
        ELSE
                byte ←UnsignedExtract8(operand3, index);
        result1 ←result1 ∨  MaskAndShift8( byte, j);
}
```

```
RDEST ←Register(result1);
```

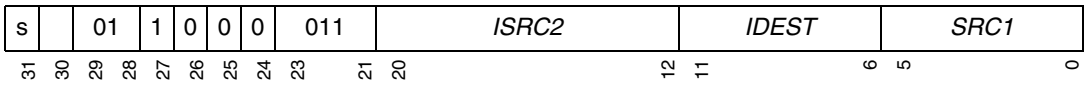
Description: Dynamic extract right operation.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

extract Immediate

extract R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend9(Imm(ISRC2));
position ←UnsignedExtract5(operand2, 0);
bitCount ←ZeroExtend4(operand2 >> 5) + 1;
IF ((bitCount + position) > 32)
        THROW ILL_INST;
sign ←(1 << (bitCount - 1));
mask ←(1 << (bitCount - 1)) - 1;
result1 ←(operand1 >> position);
IF (result1 ∧ sign)
        result1 ←result1 ∨  (~mask);
ELSE
        result1 ←result1 ∧ mask;
```

```
RIDEST ←Register(result1);
```

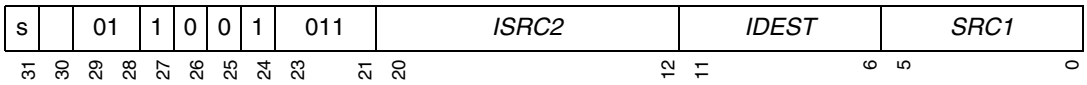
Description: Generalised signed bit field extract for small fields (1-16 bits).

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: ILL_INST

extractl Immediate

extractl R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend9(Imm(ISRC2));
position ←UnsignedExtract5(operand2, 0);
bitCount ←ZeroExtend4(operand2 >> 5) + 17;
IF ((bitCount + position) > 32)
        THROW ILL_INST;
sign ←(1 << (bitCount - 1));
mask ←(1 << (bitCount - 1)) - 1;
result1 ←(operand1 >> position);
IF (result1 ∧ sign)
        result1 ←result1 ∨  (~mask);
ELSE
        result1 ←result1 ∧ mask;
```

```
RIDEST ←Register(result1);
```

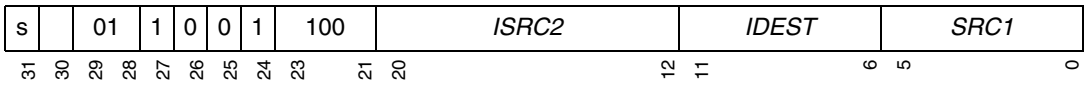
Description: Generalized signed bit field extract for large fields (>= 17 bits).

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: ILL_INST

extractlu Immediate

extractlu R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend9(Imm(ISRC2));
position ←UnsignedExtract5(operand2, 0);
bitCount ←ZeroExtend4(operand2 >> 5) + 17;
IF ((bitCount + position) > 32)
        THROW ILL_INST;
mask ←(1 << bitCount) - 1;
result1 ←(operand1 >> position) ∧ mask;
```

```
RIDEST ←Register(result1);
```

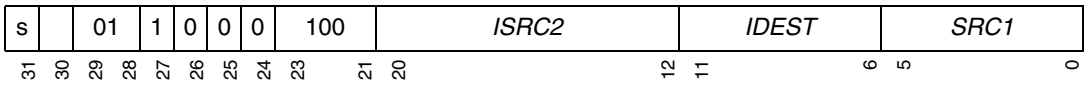
Description: Generalized unsigned bit field extract for large fields (>= 17 bits).

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: ILL_INST

extractu Immediate

extractu R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend9(Imm(ISRC2));
position ←UnsignedExtract5(operand2, 0);
bitCount ←ZeroExtend4(operand2 >> 5) + 1;
IF ((bitCount + position) > 32)
        THROW ILL_INST;
mask ←(1 << bitCount) - 1;
result1 ←(operand1 >> position) ∧ mask;
```

```
RIDEST ←Register(result1);
```

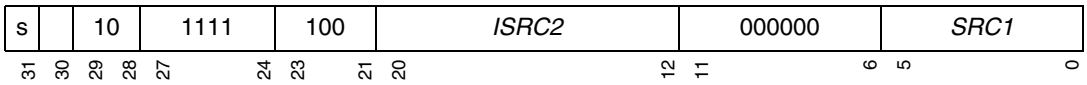
Description: Generalized unsigned bit field extract for small fields (1 - 16 bits).

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: ILL_INST

flushadd

flushadd ISRC2[R_{SRC1}]



Semantics:

operand1 ←$SignExtend_{32}$(Imm(ISRC2));
operand2 ←$SignExtend_{32}$(R_{SRC1});
ea ←$ZeroExtend_{32}$(operand1 + operand2);
PurgeAddressCheckMemory(ea);

FlushAddress(ea);

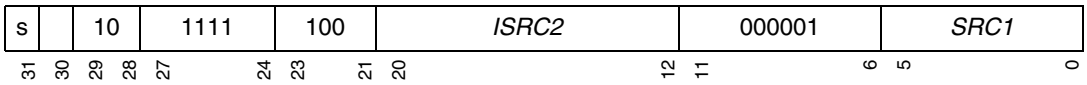
Description: Flush the address given in the L1 data memory subsystem and L2 cache (if present).

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: DTLB

flushadd.l1

flushadd.l1 ISRC2[R_{SRC1}]



Semantics:

operand1 ←SignExtend$_{32}$(Imm(ISRC2));
operand2 ←SignExtend$_{32}$(R_{SRC1});
ea ←ZeroExtend$_{32}$(operand1 + operand2);
PurgeAddressCheckMemory(ea);

FlushAddressL1(ea);

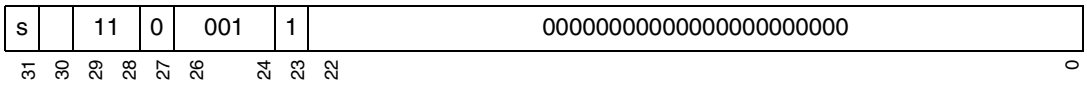
Description: Flush the address given in the L1 data memory subsystem only.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: DTLB

goto Link Register

goto $r63



Semantics:

PC ←Register(ZeroExtend$_{32}$(LR));

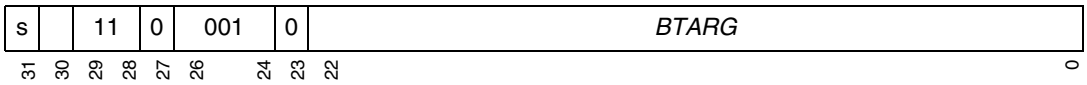
Description: Jump (using Link Register).

Restrictions: Must be the first syllable of a bundle.
There are no latency constraints between a call updating the LR and this operation.
There is a latency of 3 cycles between a load writing to the LR and this operation.
There is a latency of 2 cycles between any other operation updating the LR and this operation.

Exceptions: None.

goto Immediate

goto BTARG



Semantics:

operand1 ←SignExtend$_{23}$(BTARG)<< 2;
PC ←Register(ZeroExtend$_{32}$(BUNDLE_PC) + operand1);

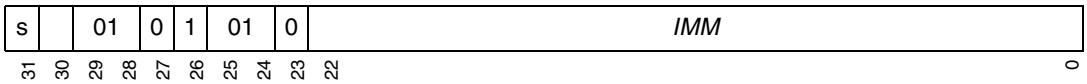
Description: Jump.

Restrictions: Must be the first syllable of a bundle.
No latency constraints.

Exceptions: None.

imml

imml IMM



Semantics:

extension ←ZeroExtend$_{23}$(IMM);

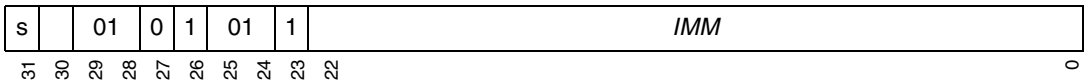
Description: Long immediate for previous syllable.

Restrictions: Must be encoded at an even word address.
No latency constraints.

Exceptions: None.

immr

immr IMM



Semantics:

extension ←ZeroExtend$_{23}$(IMM);

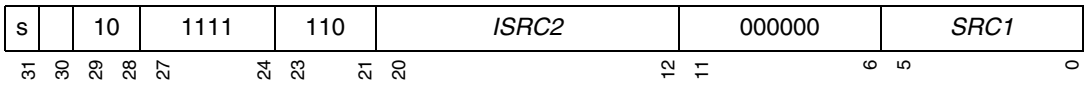
Description: Long immediate for next syllable.

Restrictions: Must be encoded at an even word address.
No latency constraints.

Exceptions: None.

invadd

invadd ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
PurgeAddressCheckMemory(ea);
```

```
InvalidateAddress(ea);
```

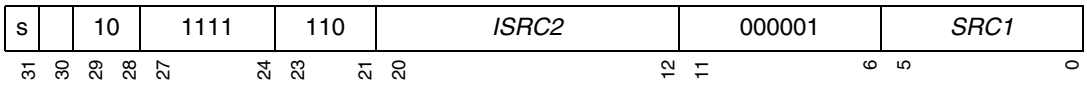
Description: Invalidate the address given in the L1 data memory subsystem and L2 cache (if present).

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: DTLB

invadd.l1

invadd.l1 ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
PurgeAddressCheckMemory(ea);
```

```
InvalidateAddressL1(ea);
```

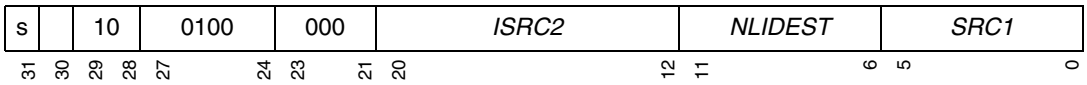
Description: Invalidate the address given in the L1 data memory subsystem only.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: DTLB

ldb

ldb $R_{NLIDEST}$ = ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakLoadHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
{
        THROW CREG_ACCESS_VIOLATION;
}
ELSE
{
        ReadCheckMemory8(ea);
}
```

```
ReadMemory8(ea);
result1 ←SignExtend8(ReadMemResponse());
RNLIDEST ←Register(result1);
```

Description: Signed load byte.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB

ldbc

ldbc $R_{NLIDEST}$ = B_{PCOND}, ISRC2[R_{SRC1}]

Semantics:

```
operand1 ←ZeroExtend4(BPCOND);
operand2 ←SignExtend32(Imm(ISRC2));
operand3 ←SignExtend32(RSRC1);
IF (operand1 ≠ 0)
{
        ea ←ZeroExtend32(operand2 + operand3);
        IF (IsDBreakLoadHit(ea))
                THROW DBREAK;
        IF (IsCRegSpace(ea))
        {
                THROW CREG_ACCESS_VIOLATION;
        }
        ELSE
        {
                ReadCheckMemory8(ea);
        }
}
```

```
IF (operand1 ≠ 0)
        ReadMemory8(ea);
IF (operand1 ≠ 0)
        result1 ←SignExtend8(ReadMemResponse());
IF (operand1)
        RNLIDEST ←Register(result1);
```

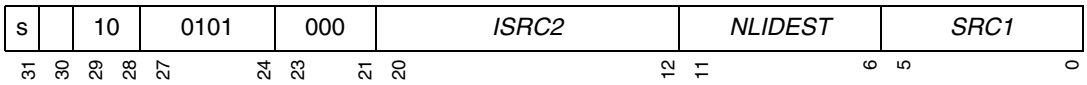
Description: Conditional signed load byte.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
B_{PCOND} can be any branch register except zero.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB

ldbu

ldbu $R_{NLIDEST}$ **=** ISRC2**[**R_{SRC1}**]**



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakLoadHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
{
        THROW CREG_ACCESS_VIOLATION;
}
ELSE
{
        ReadCheckMemory8(ea);
}
```

```
ReadMemory8(ea);
result1 ←ZeroExtend8(ReadMemResponse());
RNLIDEST ←Register(result1);
```

Description: Unsigned load byte.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB

ldbuc

ldbuc $R_{NLIDEST}$ = B_{PCOND}, ISRC2[R_{SRC1}]

Semantics:

```
operand1 ←ZeroExtend4(BPCOND);
operand2 ←SignExtend32(Imm(ISRC2));
operand3 ←SignExtend32(RSRC1);
IF (operand1 ≠ 0)
{
        ea ←ZeroExtend32(operand2 + operand3);
        IF (IsDBreakLoadHit(ea))
                THROW DBREAK;
        IF (IsCRegSpace(ea))
        {
                THROW CREG_ACCESS_VIOLATION;
        }
        ELSE
        {
                ReadCheckMemory8(ea);
        }
}
```

```
IF (operand1 ≠ 0)
        ReadMemory8(ea);
IF (operand1 ≠ 0)
        result1 ←ZeroExtend8(ReadMemResponse());
IF (operand1)
        RNLIDEST ←Register(result1);
```

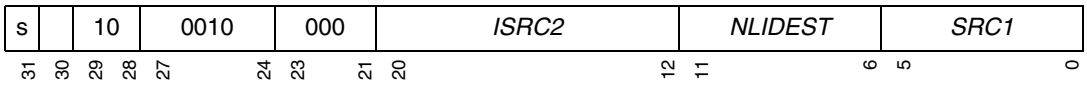
Description: Conditional unsigned load byte.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
B_{PCOND} can be any branch register except zero.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB

ldh

ldh $R_{NLIDEST}$ = ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakLoadHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
{
        THROW CREG_ACCESS_VIOLATION;
}
ELSE
{
        ReadCheckMemory16(ea);
}
```

```
ReadMemory16(ea);
result1 ←SignExtend16(ReadMemResponse());
RNLIDEST ←Register(result1);
```

Description: Signed load half-word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

ldhc

ldhc $R_{NLIDEST}$ = B_{PCOND}, ISRC2[R_{SRC1}]

Semantics:

```
operand1 ←ZeroExtend4(BPCOND);
operand2 ←SignExtend32(Imm(ISRC2));
operand3 ←SignExtend32(RSRC1);
IF (operand1 ≠ 0)
{
        ea ←ZeroExtend32(operand2 + operand3);
        IF (IsDBreakLoadHit(ea))
                THROW DBREAK;
        IF (IsCRegSpace(ea))
        {
                THROW CREG_ACCESS_VIOLATION;
        }
        ELSE
        {
                ReadCheckMemory16(ea);
        }
}
```

```
IF (operand1 ≠ 0)
        ReadMemory16(ea);
IF (operand1 ≠ 0)
        result1 ←SignExtend16(ReadMemResponse());
IF (operand1)
        RNLIDEST ←Register(result1);
```

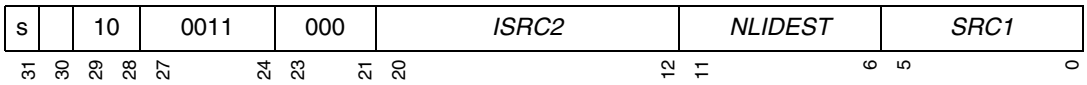
Description: Conditional signed load half-word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
B_{PCOND} can be any branch register except zero.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

ldhu

ldhu $R_{NLIDEST}$ **=** ISRC2**[**R_{SRC1}**]**



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakLoadHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
{
        THROW CREG_ACCESS_VIOLATION;
}
ELSE
{
        ReadCheckMemory16(ea);
}
```

```
ReadMemory16(ea);
result1 ←ZeroExtend16(ReadMemResponse());
RNLIDEST ←Register(result1);
```

Description: Unsigned load half-word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

ldhuc

ldhuc $R_{NLIDEST}$ = B_{PCOND}, ISRC2[R_{SRC1}]

Semantics:

```
operand1 ←ZeroExtend4(BPCOND);
operand2 ←SignExtend32(Imm(ISRC2));
operand3 ←SignExtend32(RSRC1);
IF (operand1 ≠ 0)
{
        ea ←ZeroExtend32(operand2 + operand3);
        IF (IsDBreakLoadHit(ea))
                THROW DBREAK;
        IF (IsCRegSpace(ea))
        {
                THROW CREG_ACCESS_VIOLATION;
        }
        ELSE
        {
                ReadCheckMemory16(ea);
        }
}
```

```
IF (operand1 ≠ 0)
        ReadMemory16(ea);
IF (operand1 ≠ 0)
        result1 ←ZeroExtend16(ReadMemResponse());
IF (operand1)
        RNLIDEST ←Register(result1);
```

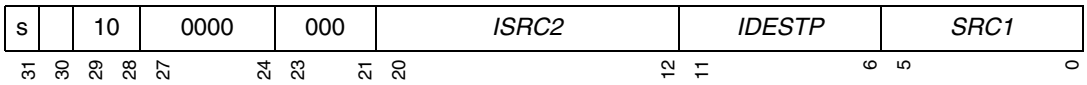
Description: Conditional unsigned load half-word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
B_{PCOND} can be any branch register except zero.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

ldl

ldl P_{IDESTP} = ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakLoadHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
{
        THROW CREG_ACCESS_VIOLATION;
}
ELSE
{
        ReadCheckMemory64(ea);
}
```

```
ReadMemory64(ea);
result1 ←SignExtend64(ReadMemResponse64());
IF (IDESTP ≠ 0)
{
        RIDESTP ←Register(result1);
        R(IDESTP + 1) ←Register(result1 >> 32);
}
```

Description: Load double word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
P_{IDESTP} can be any register pair except zero.
There is a latency of 2 cycles before P_{IDESTP} is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

ldlc

ldlc P_{IDESTP} **=** B_{PCOND}, ISRC2[R_{SRC1}]

Semantics:

```
operand1 ←ZeroExtend4(BPCOND);
operand2 ←SignExtend32(Imm(ISRC2));
operand3 ←SignExtend32(RSRC1);
IF (operand1 ≠ 0)
{
        ea ←ZeroExtend32(operand2 + operand3);
        IF (IsDBreakLoadHit(ea))
                THROW DBREAK;
        IF (IsCRegSpace(ea))
        {
                THROW CREG_ACCESS_VIOLATION;
        }
        ELSE
        {
                ReadCheckMemory64(ea);
        }
}
```

```
IF (operand1 ≠ 0)
        ReadMemory64(ea);
IF (operand1 ≠ 0)
        result1 ←SignExtend64(ReadMemResponse64());
IF (operand1)
{
        IF (IDESTP ≠ 0)
        {
                RIDESTP ←Register(result1);
                R(IDESTP + 1) ←Register(result1 >> 32);
        }
}
```

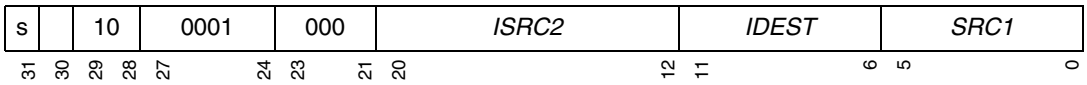
Description: Conditional load double word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
B_{PCOND} can be any branch register except zero.
P_{IDESTP} can be any register pair except zero.
There is a latency of 2 cycles before P_{IDESTP} is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

ldw

ldw R_{IDEST} = ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakLoadHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        ReadCheckCReg(ea);
ELSE
        ReadCheckMemory32(ea);
```

```
IF (IsCRegSpace(ea))
        ReadCReg(ea);
ELSE
        ReadMemory32(ea);
result1 ←SignExtend32(ReadMemResponse());
RIDEST ←Register(result1);
```

Description: Load word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
There is a latency of 2 cycles before R_{IDEST} is available for reading.
If writing the LR, there is a latency of 3 cycles before a call LR or goto LR is issued.

Exceptions: DBREAK, DTLB, CREG_ACCESS_VIOLATION, CREG_NO_MAPPING, MISALIGNED_TRAP

ldwc

ldwc R_{IDEST} = B_{PCOND}, ISRC2[R_{SRC1}]

Semantics:

```
operand1 ←ZeroExtend4(BPCOND);
operand2 ←SignExtend32(Imm(ISRC2));
operand3 ←SignExtend32(RSRC1);
IF (operand1 ≠ 0)
{
        ea ←ZeroExtend32(operand2 + operand3);
        IF (IsDBreakLoadHit(ea))
                THROW DBREAK;
        IF (IsCRegSpace(ea))
                ReadCheckCReg(ea);
        ELSE
                ReadCheckMemory32(ea);
}
```

```
IF (operand1 ≠ 0)
{
        IF (IsCRegSpace(ea))
                ReadCReg(ea);
        ELSE
                ReadMemory32(ea);
}
IF (operand1 ≠ 0)
        result1 ←SignExtend32(ReadMemResponse());
IF (operand1)
        RIDEST ←Register(result1);
```

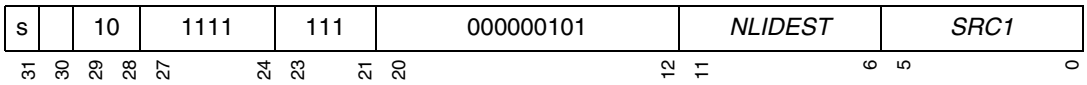
Description: Conditional load word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
B_{PCOND} can be any branch register except zero.
There is a latency of 2 cycles before R_{IDEST} is available for reading.
If writing the LR, there is a latency of 3 cycles before a call LR or goto LR is issued.

Exceptions: DBREAK, DTLB, CREG_ACCESS_VIOLATION, CREG_NO_MAPPING, MISALIGNED_TRAP

ldwl

ldwl $R_{NLIDEST}$ **=** $[R_{SRC1}]$



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
ATOMIC_LOCK←0x3;
IF (IsDBreakLoadHit(operand1))
        THROW DBREAK;
IF (IsCRegSpace(operand1))
        THROW CREG_ACCESS_VIOLATION;
ReadCheckMemory32(operand1);
ATOMIC_ADDRESS←Translate(operand1) ∧ 0xFFFFFFE0;
```

```
ReadMemory32(operand1);
result1 ←SignExtend32(ReadMemResponse());
RNLIDEST ←Register(result1);
```

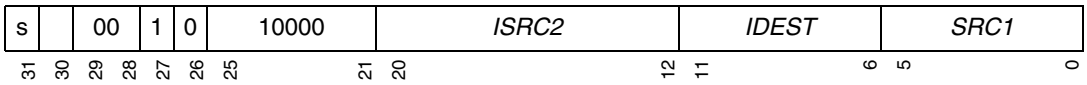
Description: Load word linked. Forms part of atomic read/modify/write sequence with stwl.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

max Immediate

max R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(Imm(ISRC2));
IF (operand1 > operand2)
        result1 ←operand1;
ELSE
        result1 ←operand2;
```

```
RIDEST ←Register(result1);
```

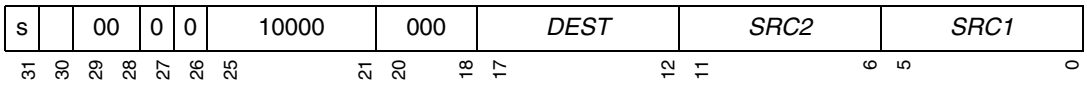
Description: Signed maximum.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

max Register

max R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
IF (operand1 > operand2)
        result1 ←operand1;
ELSE
        result1 ←operand2;
```

```
RDEST ←Register(result1);
```

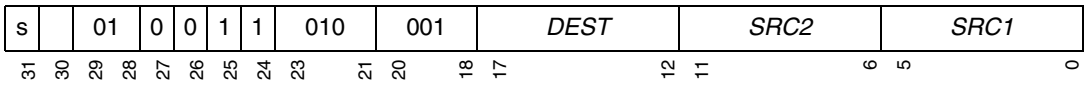
Description: Signed maximum.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

max.ph Register

max.ph R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←SignedExtract16(operand1, j);
        unpacked_opd2←SignedExtract16(operand2, j);
        IF (unpacked_opd1> unpacked_opd2)
                result1 ←result1 ∨  MaskAndShift16(unpacked_opd1, j);
        ELSE
                result1 ←result1 ∨  MaskAndShift16(unpacked_opd2, j);
}
```

```
RDEST ←Register(result1);
```

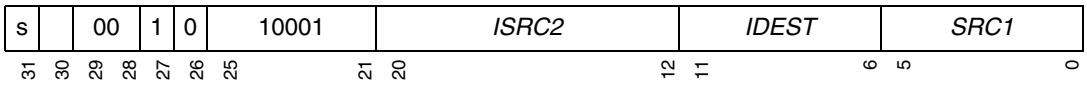
Description: Packed 16-bit signed maximum. Operands may be signed 16-bit integers or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

maxu Immediate

maxu R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(Imm(ISRC2));
IF (operand1 > operand2)
        result1 ←operand1;
ELSE
        result1 ←operand2;
```

```
RIDEST ←Register(result1);
```

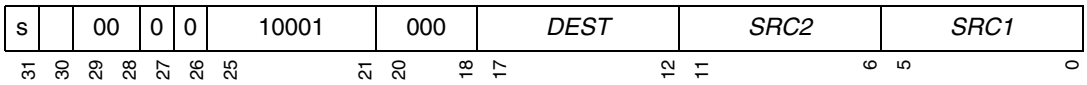
Description: Unsigned maximum.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

maxu Register

maxu R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
IF (operand1 > operand2)
        result1 ←operand1;
ELSE
        result1 ←operand2;
```

```
RDEST ←Register(result1);
```

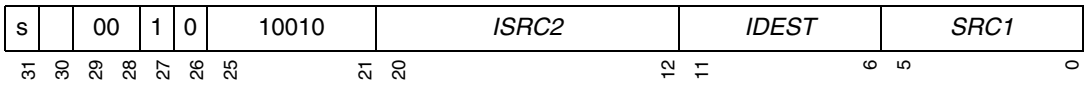
Description: Unsigned maximum.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

min Immediate

min R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(Imm(ISRC2));
IF (operand1 < operand2)
        result1 ←operand1;
ELSE
        result1 ←operand2;
```

```
RIDEST ←Register(result1);
```

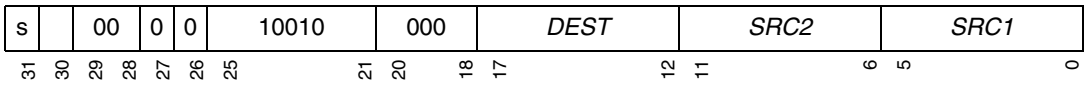
Description: Signed minimum.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

min Register

min R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
IF (operand1 < operand2)
        result1 ←operand1;
ELSE
        result1 ←operand2;
```

```
RDEST ←Register(result1);
```

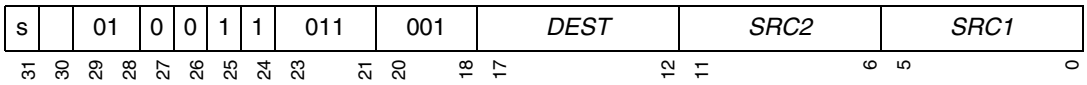
Description: Signed minimum.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

min.ph Register

min.ph R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←SignedExtract16(operand1, j);
        unpacked_opd2←SignedExtract16(operand2, j);
        IF (unpacked_opd1< unpacked_opd2)
                result1 ←result1 ∨  MaskAndShift16(unpacked_opd1, j);
        ELSE
                result1 ←result1 ∨  MaskAndShift16(unpacked_opd2, j);
}
```

```
RDEST ←Register(result1);
```

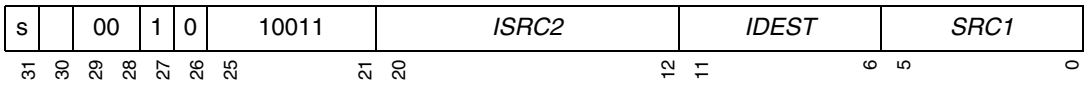
Description: Packed 16-bit signed minimum. Operands may be signed 16-bit integers or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

minu Immediate

minu R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(Imm(ISRC2));
IF (operand1 < operand2)
        result1 ←operand1;
ELSE
        result1 ←operand2;
```

```
RIDEST ←Register(result1);
```

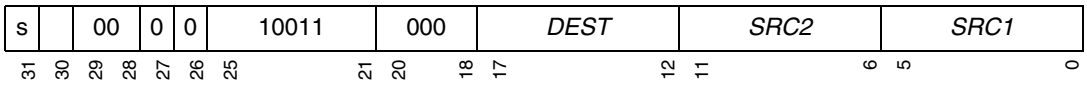
Description: Unsigned minimum.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

minu Register

minu R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
IF (operand1 < operand2)
        result1 ←operand1;
ELSE
        result1 ←operand2;
```

```
RDEST ←Register(result1);
```

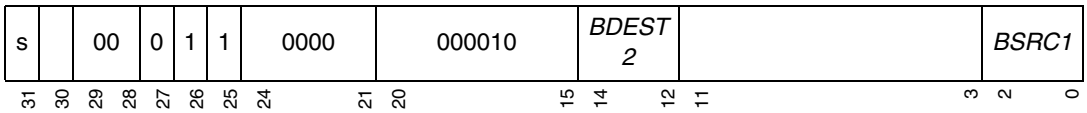
Description: Unsigned minimum.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

mov Branch Register - Branch Register

mov B_{BDEST2} = B_{BSRC1}



Semantics:

operand1 ←ZeroExtend$_4$(B_{BSRC1});
result1 ←operand1;

B_{BDEST2} ←BranchRegister(result1);

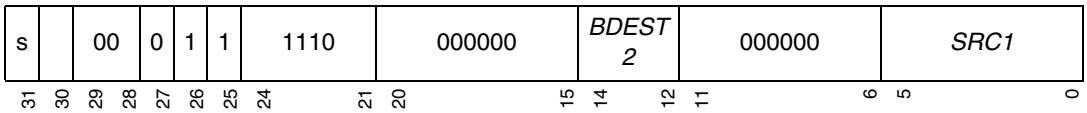
Description: Move a branch register value into branch register.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

mov Branch Register - Register

mov B_{BDEST2} = R_{SRC1}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
result1 ←ZeroExtend$_{4}$(operand1);

B_{BDEST2} ←BranchRegister(result1);

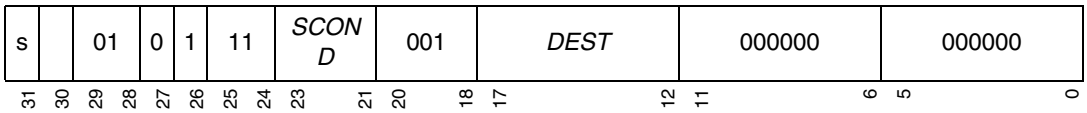
Description: Move general register value into a branch register.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

mov Register - Branch Register

mov R_{DEST} **=** B_{SCOND}



Semantics:

operand1 ←ZeroExtend$_4$(B_{SCOND});
result1 ←operand1;

R_{DEST} ←Register(result1);

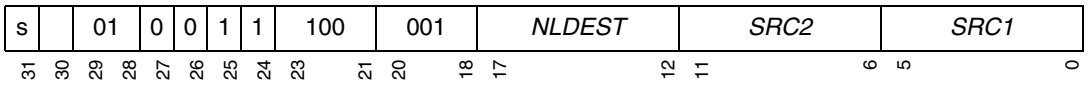
Description: Move branch register value into a general purpose register.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

mul.ph Register

mul.ph R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});

result1 ←0;
REPEAT j FROM 0 FOR 2 {
    mulresult ←SignedExtract$_{16}$(operand1, j) × SignedExtract$_{16}$(operand2, j);
    result1 ←result1 ∨ MaskAndShift$_{16}$(mulresult, j);
}
R_{NLDEST} ←Register(result1);

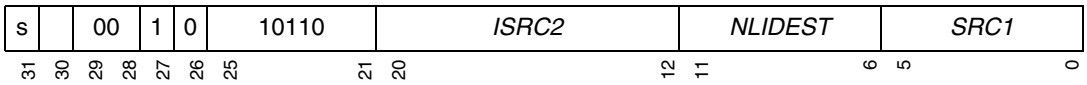
Description: Packed 16-bit multiplication. Operands may be signed or unsigned 16-bit integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mul32 Immediate

mul32 $R_{NLIDEST}$ **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));

result1 ←operand1 × operand2;
$R_{NLIDEST}$ ←Register(result1);

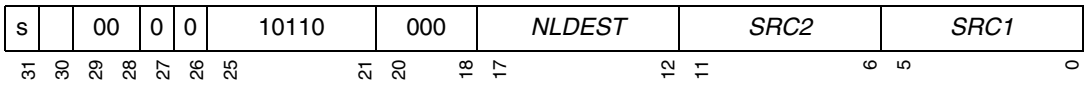
Description: 32x32-bit signed multiplication, returns lower 32 bits of the intermediate 64bit result. Operands are signed or unsigned integers.

Restrictions: Must be encoded at an odd word address.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mul32 Register

mul32 R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});

result1 ←operand1 × operand2;
R_{NLDEST} ←Register(result1);

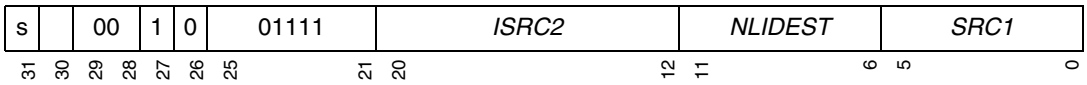
Description: 32x32-bit signed multiplication, returns lower 32 bits of the intermediate 64bit result. Operands are signed or unsigned integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mul64h Immediate

mul64h $R_{NLIDEST}$ **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));

result1 ←(operand1 × operand2) >> 32;
$R_{NLIDEST}$ ←Register(result1);

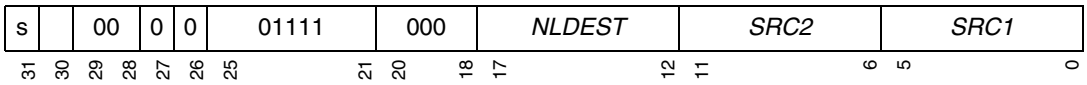
Description: 32x32-bit signed multiplication, returns upper 32 bits of the intermediate 64bit result. Operands are signed integers.

Restrictions: Must be encoded at an odd word address.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mul64h Register

mul64h R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});

result1 ←(operand1 × operand2) >> 32;
R_{NLDEST} ←Register(result1);

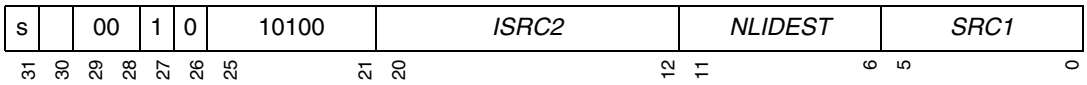
Description: 32x32-bit signed multiplication, returns upper 32 bits of the intermediate 64bit result. Operands are signed integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mul64hu Immediate

mul64hu $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(Imm(ISRC2));

result1 ←(operand1 × operand2) >> 32;
$R_{NLIDEST}$ ←Register(result1);

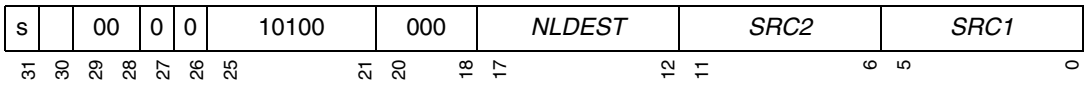
Description: 32x32-bit unsigned multiplication, returns upper 32 bits of the intermediate 64bit result. Operands are signed integers.

Restrictions: Must be encoded at an odd word address.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mul64hu Register

mul64hu R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});

result1 ←(operand1 × operand2) >> 32;
R_{NLDEST} ←Register(result1);

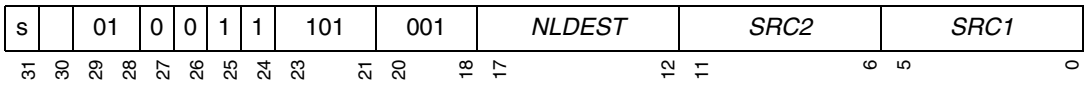
Description: 32x32-bit unsigned multiplication, returns upper 32 bits of the intermediate 64bit result. Operands are signed integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

muladd.ph Register

muladd.ph R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
```

```
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        mulresult ←SignedExtract16(operand1, j) × SignedExtract16(operand2, j);
        result1 ←result1 + mulresult;
}
result1 ←Saturate32(result1);
RNLDEST ←Register(result1);
```

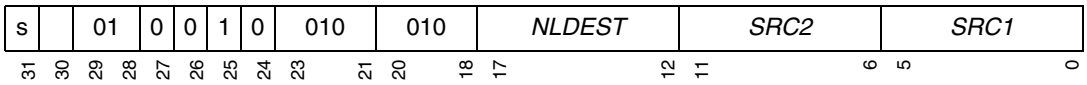
Description: Packed 16-bit signed multiplication and add across with saturation. Operands are signed 16-bit integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

muladdus.pb Register

muladdus.pb R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);

result1 ←0;
REPEAT j FROM 0 FOR 4 {
        mulresult ←UnsignedExtract8(operand1, j) × SignedExtract8(operand2, j);
        result1 ←result1 + mulresult;
}
RNLDEST ←Register(result1);
```

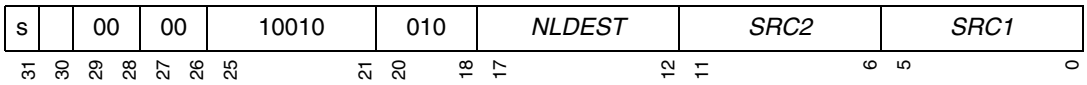
Description: Unsigned 8-bit integer multiplied by signed 8-bit integer value and add across.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mulf.n Floating point - Register

mulf.n R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});

result1 ←FMulSNonIeee(operand1, operand2);
R_{NLDEST} ←Register(result1);

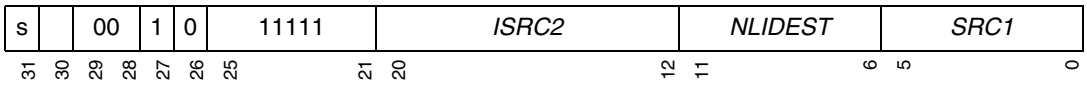
Description: IEEE754 format single precision floating point multiplication.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mulfrac Immediate

mulfrac $R_{NLIDEST}$ **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(Imm(ISRC2));

IF (((-operand1) = 0x80000000) AND ((-operand2) = 0x80000000))
{
        result1 ←0x7FFFFFFF;
}
ELSE
{
        result1 ←operand1 × operand2;
        result1 ←result1 + (1 << 30);
        result1 ←result1 >> 31;
}
RNLIDEST ←Register(result1);
```

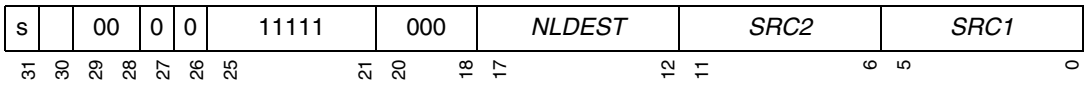
Description: 32-bit fractional multiplication with round nearest positive and saturation. Operands are fractional 1.31 format.

Restrictions: Must be encoded at an odd word address.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mulfrac Register

mulfrac R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);

IF (((-operand1) = 0x80000000) AND ((-operand2) = 0x80000000))
{
        result1 ←0x7FFFFFFF;
}
ELSE
{
        result1 ←operand1 × operand2;
        result1 ←result1 + (1 << 30);
        result1 ←result1 >> 31;
}
RNLDEST ←Register(result1);
```

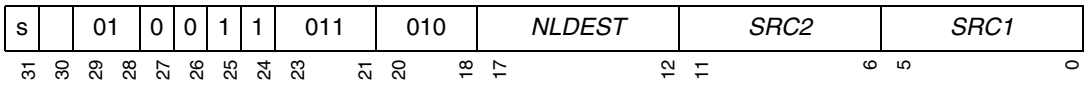
Description: 32-bit fractional multiplication with round nearest positive and saturation. Operands are fractional 1.31 format.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mulfracadds.ph Register

mulfracadds.ph R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);

result1 ←0;
REPEAT j FROM 0 FOR 2 {
        extractedoperand1 ←SignedExtract16(operand1, j);
        extractedoperand2 ←SignedExtract16(operand2, j);
        IF (((-extractedoperand1) = 0x8000) AND ((-extractedoperand2) = 0x8000))
        {
                mulresult ←0x7FFFFFFF;
        }
        ELSE
        {
                mulresult ←(extractedoperand1 × extractedoperand2) << 1;
        }
        result1 ←result1 + mulresult;
}
result1 ←Saturate32(result1);
RNLDEST ←Register(result1);
```

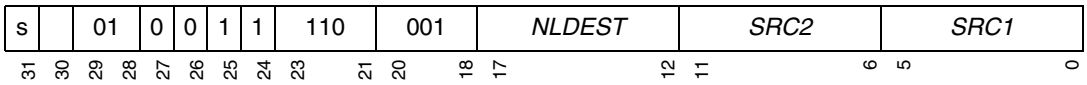
Description: Packed 16-bit signed fractional multiplication and add across with saturation. Operands are 1.15 fractional format.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mulfracrm.ph Register

mulfracrm.ph R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
```

```
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        mulresult ←SignedExtract16(operand1, j) × SignedExtract16(operand2, j);
        mulresult ←Saturate16(mulresult >> 15);
        result1 ←result1 ∨  MaskAndShift16(mulresult, j);
}
RNLDEST ←Register(result1);
```

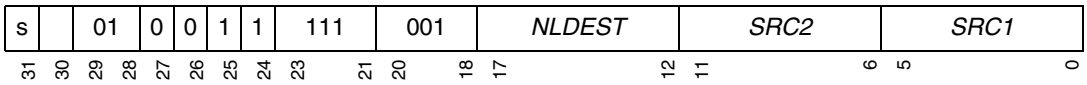
Description: Packed 16-bit fractional multiplication with round minus and saturation. Operands are fractional 1.15 format.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mulfracrne.ph Register

mulfracrne.ph R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);

result1 ←0;
REPEAT j FROM 0 FOR 2 {
        mulresult ←SignedExtract16(operand1, j) × SignedExtract16(operand2, j);
        rounding ←((1 << 14) - 1) + Bit(mulresult, 15);
        mulresult ←Saturate16((mulresult + rounding) >> 15);
        result1 ←result1 ∨  MaskAndShift16(mulresult, j);
}
RNLDEST ←Register(result1);
```

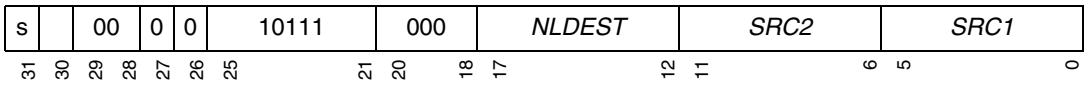
Description: Packed 16-bit fractional multiplication with round nearest even and saturation. Operands are fractional 1.15 format.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mulh Register

mulh R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});

result1 ←operand1 × (operand2 >> 16);
R_{NLDEST} ←Register(result1);

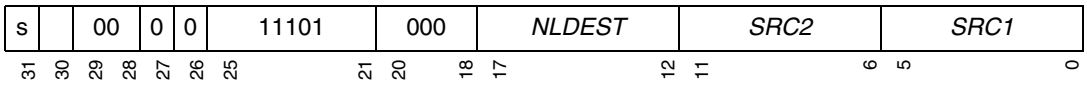
Description: Word by upper-half-word signed multiplication. Operands are 16-bit and 32-bit signed integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mulhh Register

mulhh R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});

result1 ←(operand1 >> 16) × (operand2 >> 16);
R_{NLDEST} ←Register(result1);

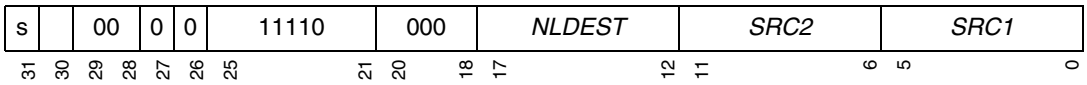
Description: Upper-half-word by upper-half-word signed multiplication. Operands are 16-bit signed integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mulhhu Register

mulhhu R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});

result1 ←ZeroExtend$_{16}$(operand1 >> 16) × ZeroExtend$_{16}$(operand2 >> 16);
R_{NLDEST} ←Register(result1);

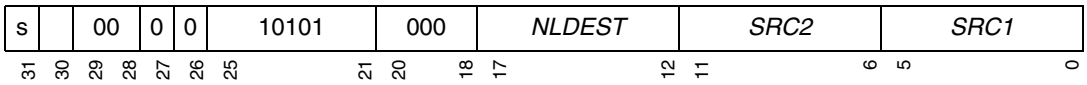
Description: Upper-half-word by upper-half-word unsigned multiplication. Operands are 16-bit unsigned integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mull Register

mull R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{16}$(R_{SRC2});

result1 ←operand1 × operand2;
R_{NLDEST} ←Register(result1);

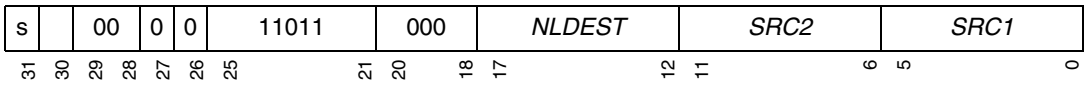
Description: Word by half-word signed multiplication. Operands are 16-bit and 32-bit unsigned integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mullh Register

mullh R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});

result1 ←operand1 × (operand2 >> 16);
R_{NLDEST} ←Register(result1);

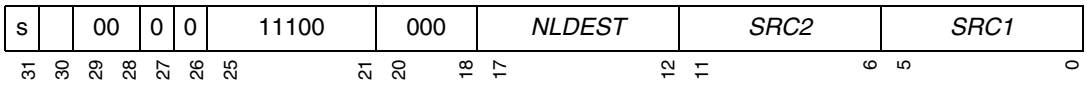
Description: Half-word by upper-half-word signed multiplication. Operands are 16-bit and 32-bit signed integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mullhu Register

mullhu R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});

result1 ←operand1 × ZeroExtend$_{16}$(operand2 >> 16);
R_{NLDEST} ←Register(result1);

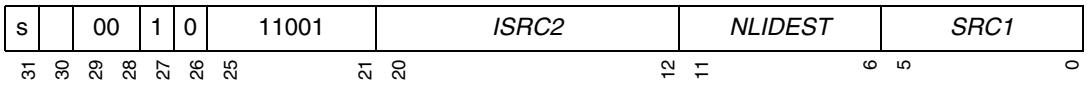
Description: Half-word by upper-half-word unsigned multiplication. Operands are 16-bit and 32-bit unsigned integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mulll Immediate

mulll $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1});
operand2 ←SignExtend$_{16}$(Imm(ISRC2));

result1 ←operand1 × operand2;
$R_{NLIDEST}$ ←Register(result1);

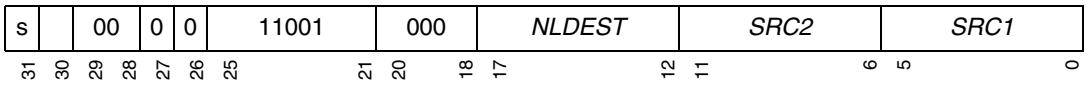
Description: Half-word by half-word signed multiplication. Operands are 16-bit signed integers.

Restrictions: Must be encoded at an odd word address.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mulll Register

mulll R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1});
operand2 ←SignExtend$_{16}$(R_{SRC2});

result1 ←operand1 × operand2;
R_{NLDEST} ←Register(result1);

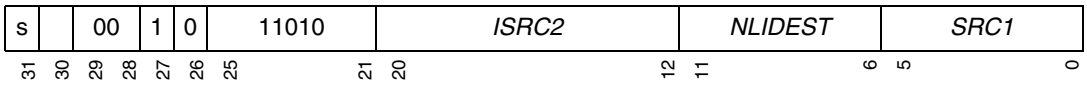
Description: Half-word by half-word signed multiplication. Operands are 16-bit signed integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

mulllu Immediate

mulllu $R_{NLIDEST}$ **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1});
operand2 ←ZeroExtend$_{16}$(Imm(ISRC2));

result1 ←operand1 × operand2;
$R_{NLIDEST}$ ←Register(result1);

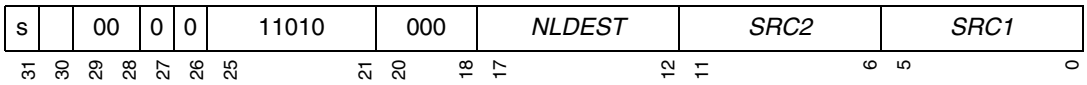
Description: Half-word by half-word unsigned multiplication. Operands are 16-bit unsigned integers.

Restrictions: Must be encoded at an odd word address.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

mulllu Register

mulllu R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1});
operand2 ←ZeroExtend$_{16}$(R_{SRC2});

result1 ←operand1 × operand2;
R_{NLDEST} ←Register(result1);

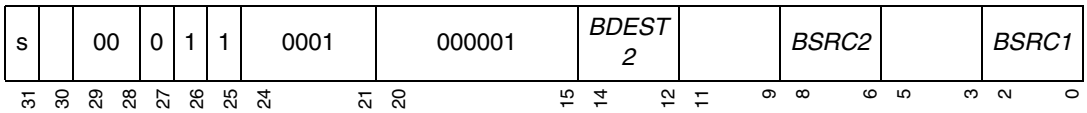
Description: Half-word by half-word unsigned multiplication. Operands are 16-bit unsigned integers.

Restrictions: Must be encoded at an odd word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

nandl Branch Register - Branch Register

nandl B_{BDEST2} = B_{BSRC1}, B_{BSRC2}



Semantics:

operand1 ←ZeroExtend$_4$(B_{BSRC1});
operand2 ←ZeroExtend$_4$(B_{BSRC2});
result1 ←NOT ((operand1 ≠ 0) AND (operand2 ≠ 0));

B_{BDEST2} ←BranchRegister(result1);

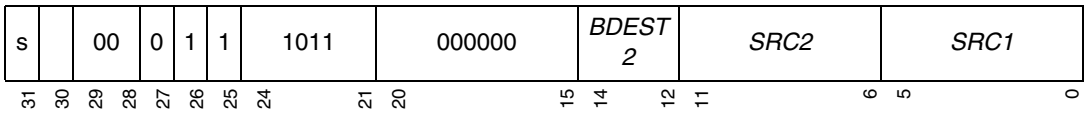
Description: Logical NAND.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

nandl Branch Register - Register

nandl B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←NOT ((operand1 ≠ 0) AND (operand2 ≠ 0));

B_{BDEST2} ←BranchRegister(result1);

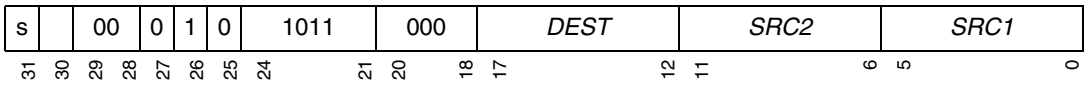
Description: Logical NAND.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

nandl Register - Register

nandl R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←NOT ((operand1 ≠ 0) AND (operand2 ≠ 0));

R_{DEST} ←Register(result1);

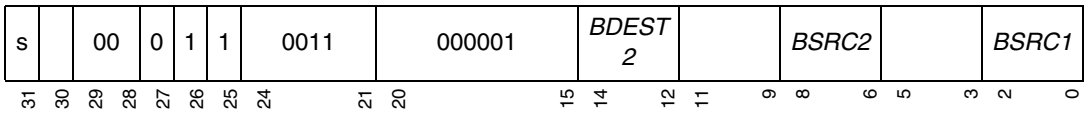
Description: Logical NAND.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

norl Branch Register - Branch Register

norl B_{BDEST2} = B_{BSRC1}, B_{BSRC2}



Semantics:

operand1 ←ZeroExtend$_4$(B_{BSRC1});
operand2 ←ZeroExtend$_4$(B_{BSRC2});
result1 ←NOT((operand1 ≠ 0) OR (operand2 ≠ 0));

B_{BDEST2} ←BranchRegister(result1);

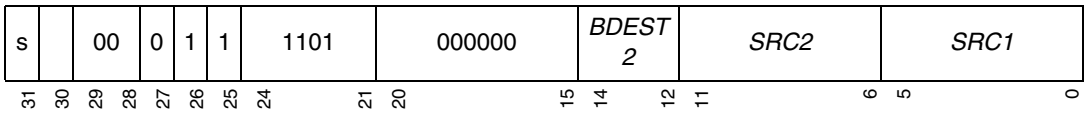
Description: Logical NOR.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

norl Branch Register - Register

norl B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
result1 ←NOT ((operand1 ≠ 0) OR (operand2 ≠ 0));
```

```
BBDEST2 ←BranchRegister(result1);
```

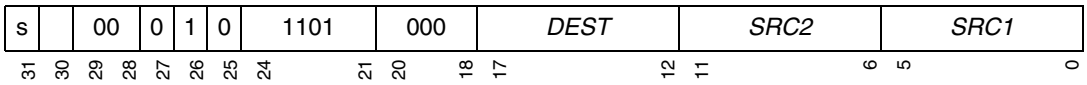
Description: Logical NOR.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

norl Register - Register

norl R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←NOT ((operand1 ≠ 0) OR (operand2 ≠ 0));

R_{DEST} ←Register(result1);

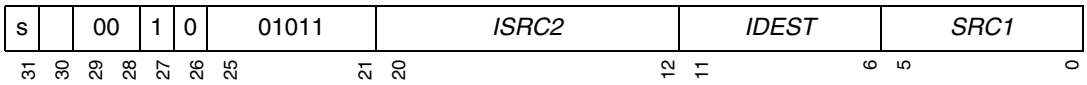
Description: Logical NOR.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

or Immediate

or R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ∨ operand2;

R_{IDEST} ←Register(result1);

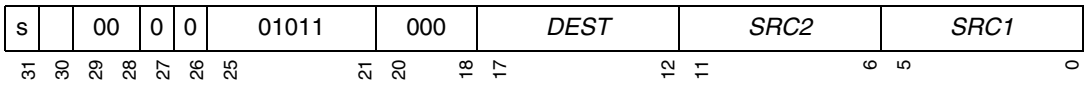
Description: Bitwise OR.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

or Register

or $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 ∨ operand2;

R_{DEST} ←Register(result1);

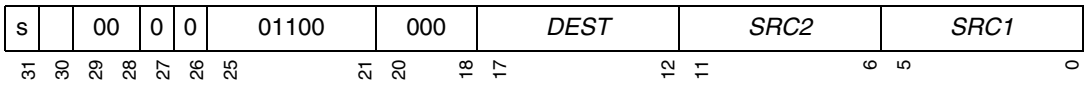
Description: Bitwise OR.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

orc Register

orc R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(~operand1) ∨ operand2;

R_{DEST} ←Register(result1);

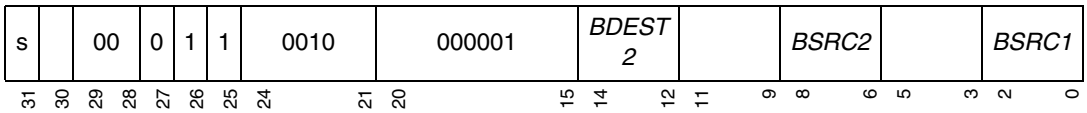
Description: Negate operand 1 and then bitwise OR.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

orl Branch Register - Branch Register

orl B_{BDEST2} = B_{BSRC1}, B_{BSRC2}



Semantics:

operand1 ←ZeroExtend$_4$(B_{BSRC1});
operand2 ←ZeroExtend$_4$(B_{BSRC2});
result1 ←(operand1 ≠ 0) OR (operand2 ≠ 0);

B_{BDEST2} ←BranchRegister(result1);

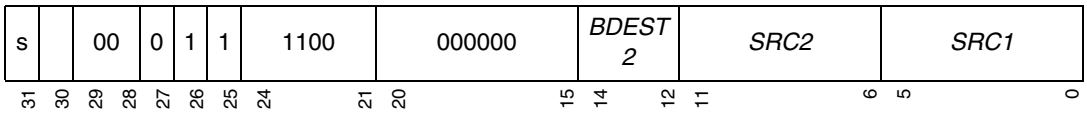
Description: Logical OR.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

orl Branch Register - Register

orl B_{BDEST2} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
result1 ←(operand1 ≠ 0) OR (operand2 ≠ 0);
```

```
BBDEST2 ←BranchRegister(result1);
```

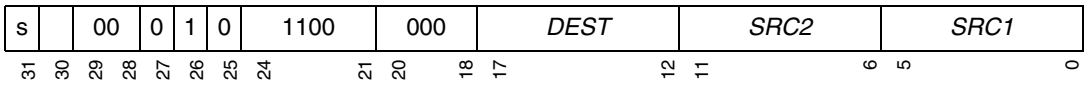
Description: Logical OR.

Restrictions: No address or bundle restrictions.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: None.

orl Register - Register

orl R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 ≠ 0) OR (operand2 ≠ 0);

R_{DEST} ←Register(result1);

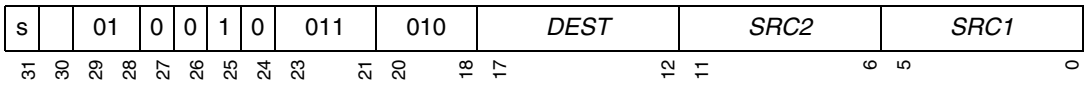
Description: Logical OR.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

pack.pb Register

pack.pb R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←SignedExtract16(operand1, j);
        unpacked_opd2←SignedExtract16(operand2, j);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd1, j);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd2, j + 2);
}
```

```
RDEST ←Register(result1);
```

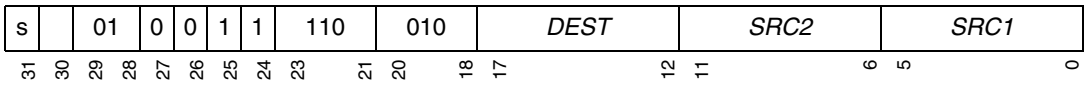
Description: Pack 4 16-bit values to 8-bit results ignoring upper bits.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

packrnp.phh Register

packrnp.phh R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
rounded_opd1←(operand1 + 0x8000) >> 16;
rounded_opd2←(operand2 + 0x8000) >> 16;
saturated_opd1←Saturate$_{16}$(rounded_opd1);
saturated_opd2←Saturate$_{16}$(rounded_opd2);
result1 ←MaskAndShift$_{16}$(saturated_opd1, 0) ∨ MaskAndShift$_{16}$(saturated_opd2, 1);

R_{DEST} ←Register(result1);

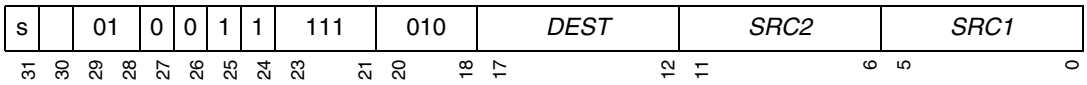
Description: Pack high part of 32-bit signed value into 16-bit signed results with round nearest positive.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

packs.ph Register

packs.ph R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
saturated_opd1←Saturate$_{16}$(operand1);
saturated_opd2←Saturate$_{16}$(operand2);
result1 ←MaskAndShift$_{16}$(saturated_opd1, 0) ∨ MaskAndShift$_{16}$(saturated_opd2, 1);

R_{DEST} ←Register(result1);

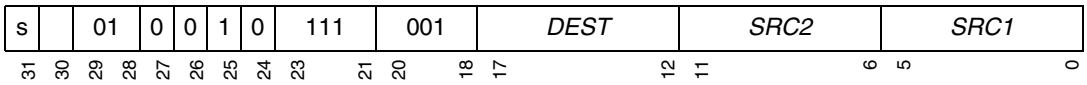
Description: Pack 32-bit signed values into 16-bit signed results with saturation.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

packsu.pb Register

packsu.pb R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←UnsignedSaturate8(SignedExtract16(operand1, j));
        unpacked_opd2←UnsignedSaturate8(SignedExtract16(operand2, j));
        result1 ←result1 ∨ MaskAndShift8(unpacked_opd1, j);
        result1 ←result1 ∨ MaskAndShift8(unpacked_opd2, j + 2);
}
```

```
RDEST ←Register(result1);
```

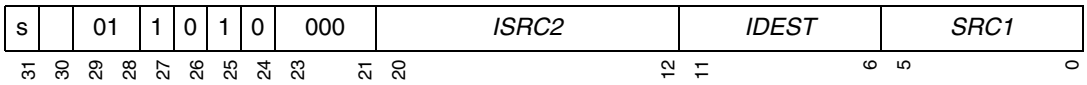
Description: Pack 16-bit signed values into 8-bit unsigned results with saturation.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

perm.pb Immediate

perm.pb R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        byteselect ←UnsignedExtract2(operand2, j);
        unpacked_opd1←UnsignedExtract8(operand1, byteselect);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd1, j);
}
```

```
RIDEST ←Register(result1);
```

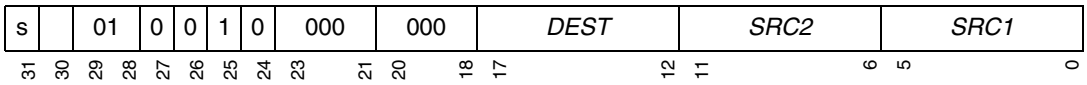
Description: Packed 8-bit permute.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

perm.pb Register

perm.pb R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        byteselect ←UnsignedExtract2(operand2, j);
        unpacked_opd1←UnsignedExtract8(operand1, byteselect);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd1, j);
}
```

```
RDEST ←Register(result1);
```

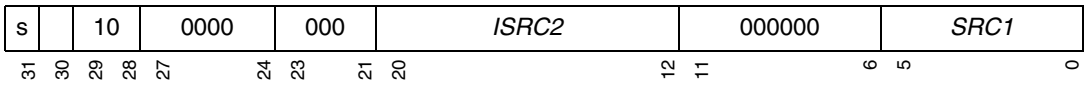
Description: Packed 8-bit permute.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

pft

pft ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakPrefetchHit(ea))
        THROW DBREAK;
PrefetchCheckMemory(ea);
```

```
PrefetchMemory(ea);
```

Description: Prefetch.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
$R_{NLIDEST}$ can be any general register except the link register.
No latency constraints.

Exceptions: DBREAK, DTLB

pftc

pftc B_{PCOND}, ISRC2[R_{SRC1}]

Semantics:

```
operand1 ←ZeroExtend4(BPCOND);
operand2 ←SignExtend32(Imm(ISRC2));
operand3 ←SignExtend32(RSRC1);
IF (operand1 ≠ 0)
{
        ea ←ZeroExtend32(operand2 + operand3);
        IF (IsDBreakPrefetchHit(ea))
                THROW DBREAK;
        PrefetchCheckMemory(ea);
}
```

```
IF (operand1 ≠ 0)
        PrefetchMemory(ea);
```

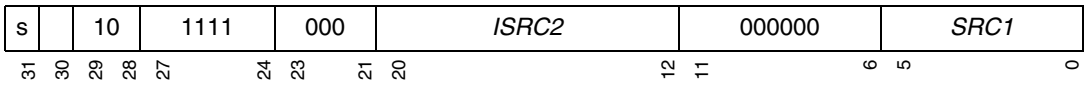
Description: Conditional prefetch.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
B_{PCOND} can be any branch register except zero.
$R_{NLIDEST}$ can be any general register except the link register.
No latency constraints.

Exceptions: DBREAK, DTLB

prgadd

prgadd ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
PurgeAddressCheckMemory(ea);
```

```
PurgeAddress(ea);
```

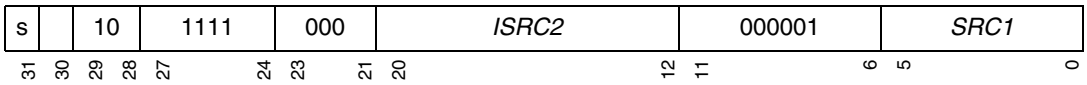
Description: Purge the address from the data memory subsystem.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: DTLB

prgadd.l1

prgadd.l1 ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
PurgeAddressCheckMemory(ea);
```

```
PurgeAddressL1(ea);
```

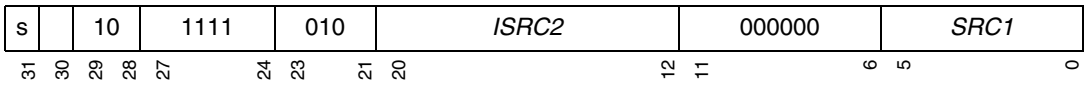
Description: Purge the address from the L1 data memory subsystem only (excluding any L2 cache).

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: DTLB

prginsadd

prginsadd ISRC2[R_{SRC1}]



Semantics:

operand1 ←$SignExtend_{32}$(Imm(ISRC2));
operand2 ←$SignExtend_{32}$(R_{SRC1});
ea ←$ZeroExtend_{32}$(operand1 + operand2);
PurgeInsAddressCheckMemory(ea);

PurgeInsAddress(ea);

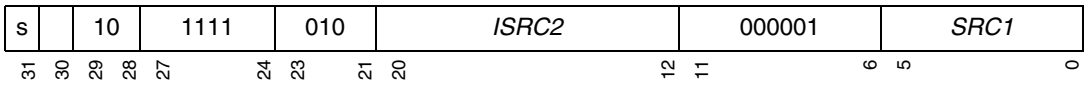
Description: Purge the address from the instruction memory subsystem.

Restrictions: Must be the only operation in a bundle.
No latency constraints.

Exceptions: ITLB

prginsadd.l1

prginsadd.l1 ISRC2[R_{SRC1}]



Semantics:

operand1 ←$SignExtend_{32}$(Imm(ISRC2));
operand2 ←$SignExtend_{32}$(R_{SRC1});
ea ←$ZeroExtend_{32}$(operand1 + operand2);
PurgeInsAddressCheckMemory(ea);

PurgeInsAddressL1(ea);

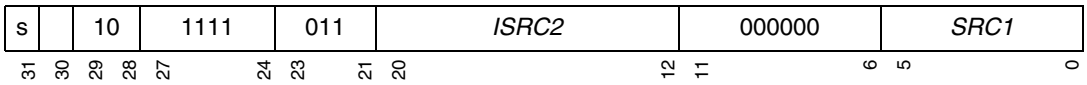
Description: Purge the address from the L1 instruction memory subsystem (excluding any L2 cache).

Restrictions: Must be the only operation in a bundle.
No latency constraints.

Exceptions: ITLB

prginsset

prginsset ISRC2[R_{SRC1}]



Semantics:

operand1 ←$SignExtend_{32}$(Imm(ISRC2));
operand2 ←$SignExtend_{32}$(R_{SRC1});
ea ←$ZeroExtend_{32}$(operand1 + operand2);

PurgeInsSet(ea);

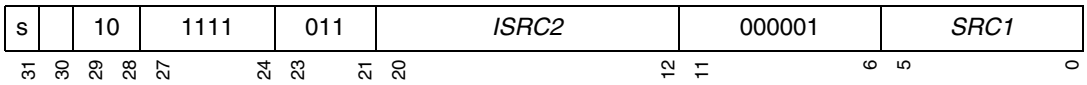
Description: Purge a set of cache lines from the instruction memory subsystem.

Restrictions: Must be the only operation in a bundle.
No latency constraints.

Exceptions: None.

prginsset.l1

prginsset.l1 ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
```

```
PurgeInsSetL1(ea);
```

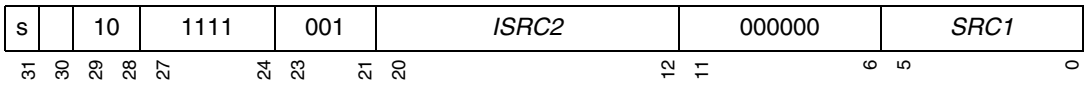
Description: Purge a set of cache lines from the L1 instruction memory subsystem only (excluding any L2 cache).

Restrictions: Must be the only operation in a bundle.
No latency constraints.

Exceptions: None.

prgset

prgset ISRC2[R_{SRC1}]



Semantics:

operand1 ←SignExtend$_{32}$(Imm(ISRC2));
operand2 ←SignExtend$_{32}$(R_{SRC1});
ea ←ZeroExtend$_{32}$(operand1 + operand2);

PurgeSet(ea);

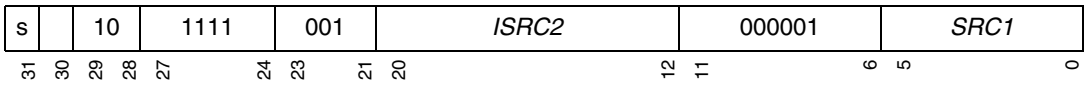
Description: Purge a set of cache lines from the data memory subsystem.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: None.

prgset.l1

prgset.l1 ISRC2[R_{SRC1}]



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
ea ←ZeroExtend32(operand1 + operand2);
```

```
PurgeSetL1(ea);
```

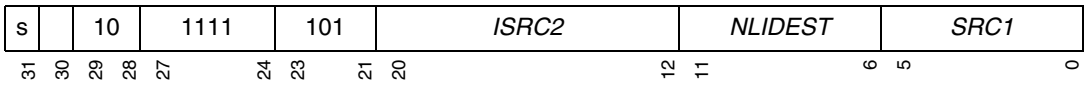
Description: Purge a set of cache lines from the L1 data memory subsystem only (excluding any L2 cache).

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: None.

pswmask

pswmask $R_{NLIDEST}$ **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
IF ((PSW[USER_MODE]) AND (NOT(PSW[DEBUG_MODE])))
        THROW ILL_INST;
```

```
result1 ←PswMask(operand1, operand2);
RNLIDEST ←Register(result1);
```

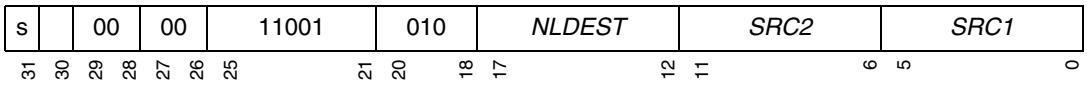
Description: Atomic psw update. The immediate value specifies a bit mask. The masked bits of the PSW are replaced by the corresponding bit of the register. Returns the unmodified PSW.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 2 cycles before $R_{NLIDEST}$ is available for reading.
There is a latency of 2 cycles between an operation writing PSW and this operation being issued.

Exceptions: ILL_INST

rem Register

rem R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});

result1 ←IRemIeee(operand1, operand2);
R_{NLDEST} ←Register(result1);

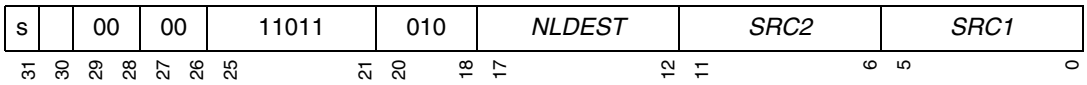
Description: Signed integer remainder. May stall the pipeline, see definition of IRemIeee(). Operands are signed integers.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

remu Register

remu R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});

result1 ←UIRemIeee(operand1, operand2);
R_{NLDEST} ←Register(result1);

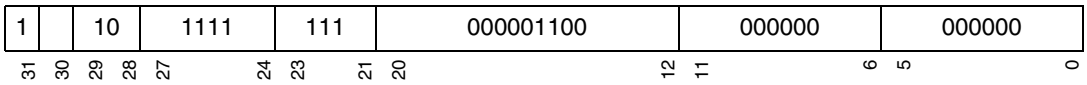
Description: Unsigned integer remainder. May stall the pipeline, see definition of UIRemIeee(). Operands are unsigned integers.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

retention

retention



Semantics:

Retention();

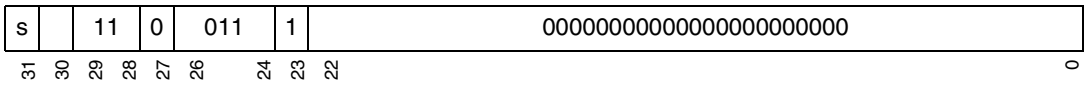
Description: Enter retention power saving mode.

Restrictions: Must be the only operation in a bundle.
No latency constraints.

Exceptions: None.

return Link Register

return $r63



Semantics:

PC ←Register(ZeroExtend$_{32}$(LR));

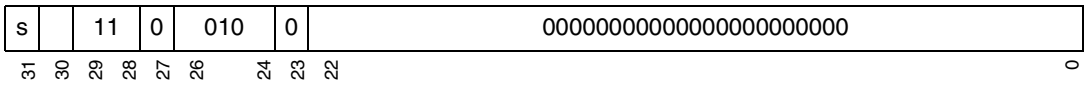
Description: Return (to Link Register).

Restrictions: Must be the first syllable of a bundle.
There are no latency constraints between a call updating the LR and this operation.
There is a latency of 3 cycles between a load writing to the LR and this operation.
There is a latency of 2 cycles between any other operation updating the LR and this operation.

Exceptions: None.

rfi

rfi



Semantics:

```
IF ((PSW[USER_MODE]) AND (NOT(PSW[DEBUG_MODE])))
        THROW ILL_INST;
PC ←Register(ZeroExtend32(SAVED_PC));
```

```
PSW ←SAVED_PSW;
SAVED_PC←SAVED_SAVED_PC;
SAVED_PSW←SAVED_SAVED_PSW;
ATOMIC_LOCK←0;
Rfi();
```

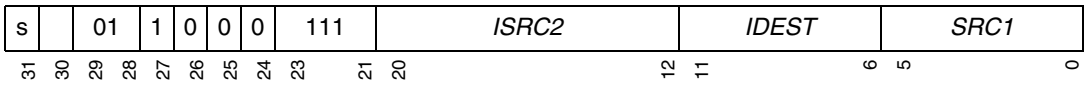
Description: Return from interrupt.

Restrictions: Must be the first in a bundle and uses the ld/st unit for which only one operation is allowed per bundle.
Operations writing SAVED_PC must be followed by 4 bundles before this operation can be issued.
Operations writing SAVED_PSW must be followed by 4 bundles before this operation can be issued.
Operations writing SAVED_SAVED_PC must be followed by 4 bundles before this operation can be issued.
Operations writing SAVED_SAVED_PSW must be followed by 4 bundles before this operation can be issued.
Operations writing PSW must be followed by 4 bundles before this operation can be issued.

Exceptions: ILL_INST

rotl Immediate

rotl R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{5}$(Imm(ISRC2));
result1 ←(operand1 << operand2) ∨ (operand1 >> (32 - operand2));

R_{IDEST} ←Register(result1);

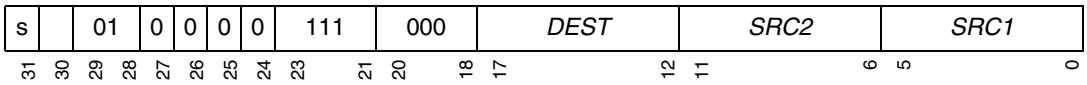
Description: Scalar 32bit rotate left.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

rotl Register

rotl R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{5}$(R_{SRC2});
result1 ←(operand1 << operand2) ∨ (operand1 >> (32 - operand2));

R_{DEST} ←Register(result1);

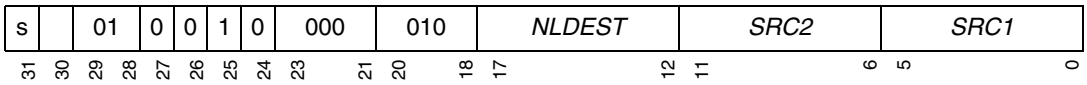
Description: Scalar 32bit rotate left.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sadu.pb Register

sadu.pb R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        subresult ←UnsignedExtract8(operand1, j) - UnsignedExtract8(operand2, j);
        IF (subresult > 0)
                result1 ←result1 + subresult;
        ELSE
                result1 ←result1 - subresult;
}
```

```
RNLDEST ←Register(result1);
```

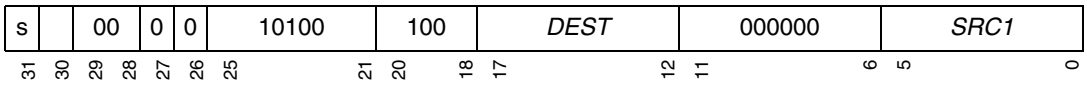
Description: Sum of absolute differences on packed unsigned 8-bit values.

Restrictions: No address or bundle restrictions.
R_{NLDEST} can be any general register except the link register.
There is a latency of 1 cycle before R_{NLDEST} is available for reading.

Exceptions: None.

sats Register

sats R_{DEST} = R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
result1 ←Saturate$_{16}$(operand1);

R_{DEST} ←Register(result1);

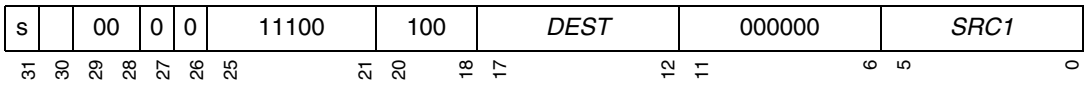
Description: Saturate from 32bit scalar to 16bit scalar. The operand is a signed integer or is fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

satso Register

satso R_{DEST} **=** R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
result1 ←Overflow$_{16}$(operand1);

R_{DEST} ←Register(result1);

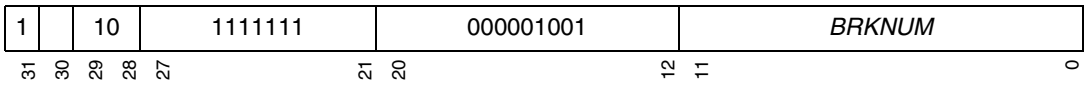
Description: Indicate if a sats operation with the same operands would saturate.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sbrk

sbrk BRKNUM



Semantics:

operand1 ←ZeroExtend$_{12}$(BRKNUM);
THROW SBREAK;

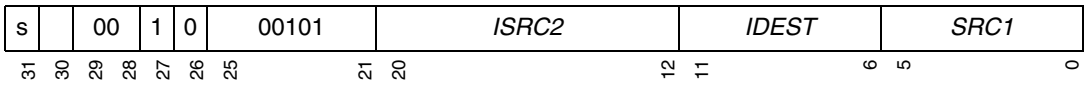
Description: Software breakpoint.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: SBREAK

sh1add Immediate

sh1add R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←(operand1 << 1) + operand2;

R_{IDEST} ←Register(result1);

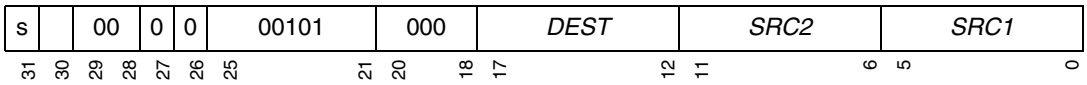
Description: Shift the first operand left one place and perform a 32bit scalar addition with the second operand. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sh1add Register

sh1add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 << 1) + operand2;

R_{DEST} ←Register(result1);

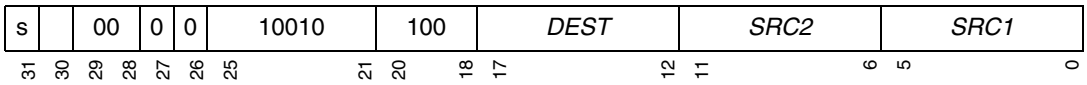
Description: Shift the first operand left one place and perform a 32bit scalar addition with the second operand. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sh1adds Register

sh1adds R_{DEST} = R_{SRC2}, R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
shiftresult ←Saturate$_{32}$(operand1 << 1);
result1 ←Saturate$_{32}$(shiftresult + operand2);

R_{DEST} ←Register(result1);

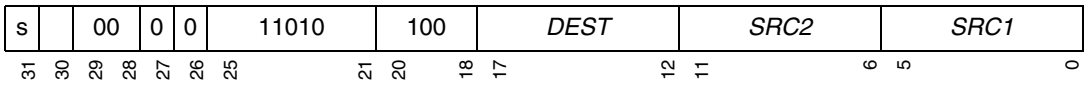
Description: Shift Rsrc1 left one place, saturate to 32 bits and then add to Rsrc2 and saturate again. Operands may be signed integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sh1addso Register

sh1addso R_{DEST} = R_{SRC2}, R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
shiftresult ←operand1 << 1;
addresult ←shiftresult + operand2;
result1 ←Overflow$_{32}$(shiftresult) OR Overflow$_{32}$(addresult);

R_{DEST} ←Register(result1);

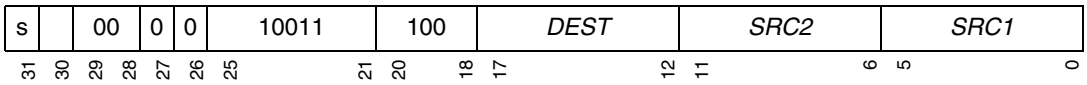
Description: Indicates whether a sh1adds operation with the given input operands causes a saturation.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sh1subs Register

sh1subs R_{DEST} = R_{SRC2}, R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
shiftresult ←Saturate$_{32}$(operand1 << 1);
result1 ←Saturate$_{32}$(operand2 - shiftresult);

R_{DEST} ←Register(result1);

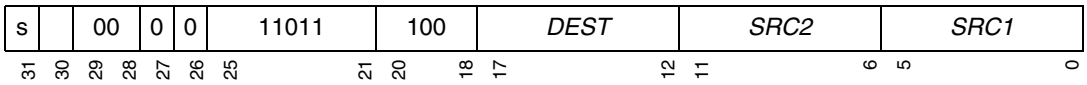
Description: Shift Rsrc1 left one place, saturate to 32 bits and then subtract from Rsrc2 and saturate again. Operands may be signed integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sh1subso Register

sh1subso R_{DEST} = R_{SRC2}, R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
shiftresult ←operand1 << 1;
subresult ←operand2 - shiftresult;
result1 ←Overflow$_{32}$(shiftresult) OR Overflow$_{32}$(subresult);

R_{DEST} ←Register(result1);

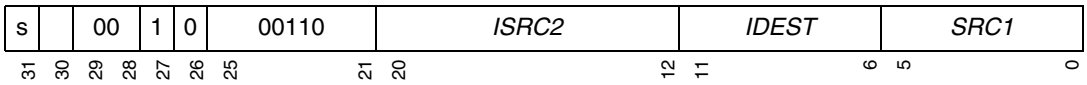
Description: Indicates whether a sh1subs operation with the given input operands causes a saturation.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sh2add Immediate

sh2add R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←(operand1 << 2) + operand2;

R_{IDEST} ←Register(result1);

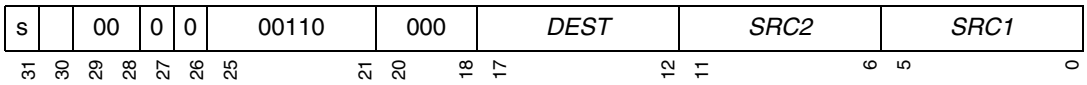
Description: Shift the first operand left two places and perform a 32bit scalar addition with the second operand. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sh2add Register

sh2add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 << 2) + operand2;

R_{DEST} ←Register(result1);

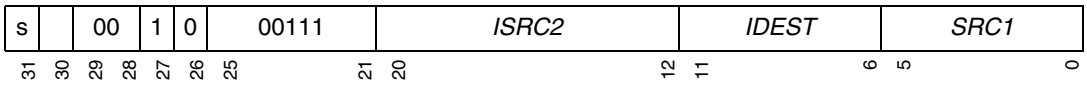
Description: Shift the first operand left two places and perform a 32bit scalar addition with the second operand. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sh3add Immediate

sh3add R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←(operand1 << 3) + operand2;

R_{IDEST} ←Register(result1);

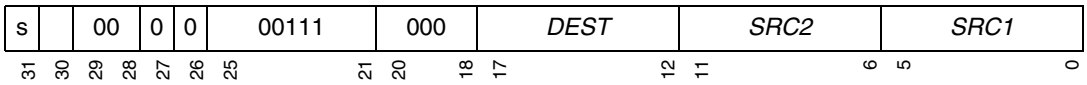
Description: Shift the first operand left three places and perform a 32bit scalar addition with the second operand. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sh3add Register

sh3add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←(operand1 << 3) + operand2;

R_{DEST} ←Register(result1);

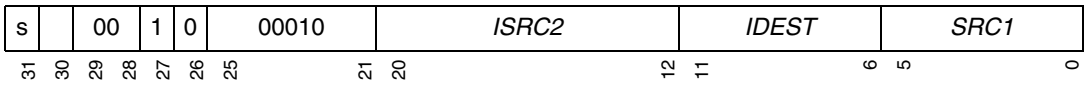
Description: Shift the first operand left three places and perform a 32bit scalar addition with the second operand. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shl Immediate

shl R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
IF (operand2 > 31)
        result1 ←0;
ELSE
        result1 ←operand1 << operand2;
```

R_{IDEST} ←Register(result1);

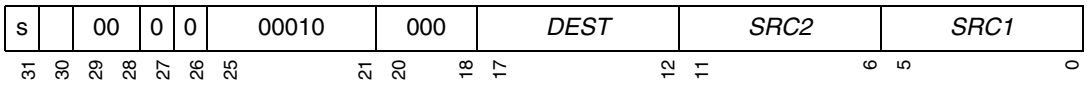
Description: Scalar 32bit arithmetic or logical left shift. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shl Register

shl R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
IF (operand2 > 31)
        result1 ←0;
ELSE
        result1 ←operand1 << operand2;
```

```
RDEST ←Register(result1);
```

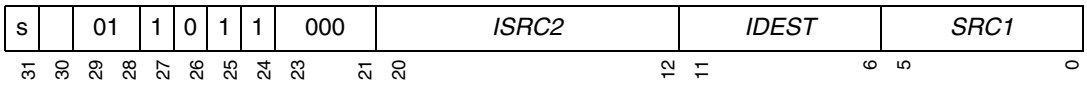
Description: Scalar 32bit arithmetic or logical left shift. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shl.ph Immediate

shl.ph R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
result1 ←0;
IF (operand2 < 16)
        REPEAT j FROM 0 FOR 2   {
        unpacked_opd1←SignedExtract16(operand1, j);
        result1 ←result1 ∨  MaskAndShift16(unpacked_opd1<< operand2, j);
}
```

R_{IDEST} ←Register(result1);

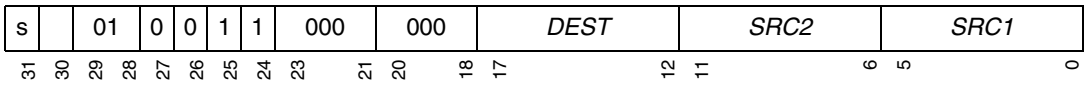
Description: Packed 16-bit left shift. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shl.ph Register

shl.ph R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
result1 ←0;
IF (operand2 < 16)
        REPEAT j FROM 0 FOR 2   {
        unpacked_opd1←SignedExtract16(operand1, j);
        result1 ←result1 ∨  MaskAndShift16(unpacked_opd1<< operand2, j);
}
```

```
RDEST ←Register(result1);
```

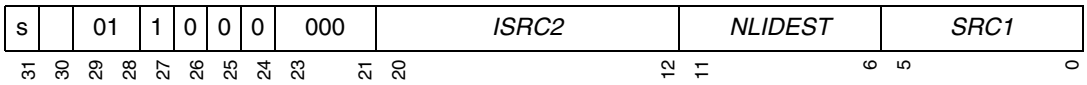
Description: Packed 16-bit left shift. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shls Immediate

shls $R_{NLIDEST}$ **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
IF (operand2 > 32)
        shiftdistance ←32;
ELSE
        shiftdistance ←operand2;
result1 ←Saturate32(operand1 << shiftdistance);
```

$R_{NLIDEST}$ ←Register(result1);

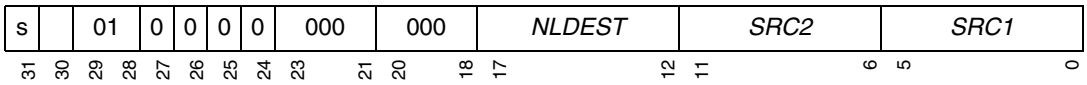
Description: Scalar 32bit left shift with saturation. Operands may be signed integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 1 cycle before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

shls Register

shls R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
IF (operand2 > 32)
        shiftdistance ←32;
ELSE
        shiftdistance ←operand2;
result1 ←Saturate32(operand1 << shiftdistance);
```

```
RNLDEST ←Register(result1);
```

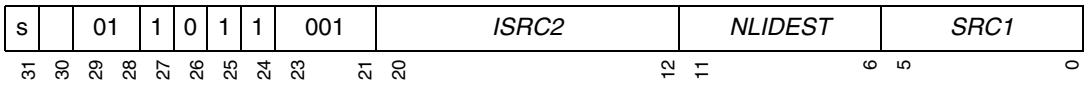
Description: Scalar 32bit left shift with saturation. Operands may be signed integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
R_{NLDEST} can be any general register except the link register.
There is a latency of 1 cycle before R_{NLDEST} is available for reading.

Exceptions: None.

shls.ph Immediate

shls.ph $R_{NLIDEST}$ **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
result1 ←0;
IF (operand2 > 16)
        shiftdistance ←16;
ELSE
        shiftdistance ←operand2;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←SignedExtract16(operand1, j);
        satresult ←Saturate16(unpacked_opd1<< shiftdistance);
        result1 ←result1 ∨  MaskAndShift16(satresult, j);
}
```

$R_{NLIDEST}$ ←Register(result1);

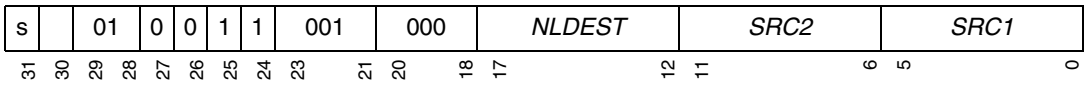
Description: Packed 16-bit signed shift left with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 1 cycle before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

shls.ph Register

shls.ph R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
result1 ←0;
IF (operand2 > 16)
        shiftdistance ←16;
ELSE
        shiftdistance ←operand2;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←SignedExtract16(operand1, j);
        satresult ←Saturate16(unpacked_opd1<< shiftdistance);
        result1 ←result1 ∨  MaskAndShift16(satresult, j);
}
```

```
RNLDEST ←Register(result1);
```

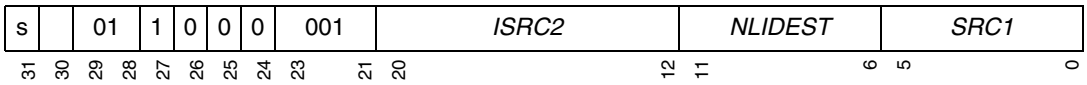
Description: Packed 16-bit signed shift left with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
R_{NLDEST} can be any general register except the link register.
There is a latency of 1 cycle before R_{NLDEST} is available for reading.

Exceptions: None.

shlso Immediate

shlso $R_{NLIDEST}$ **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
IF (operand2 > 32)
        shiftdistance ←32;
ELSE
        shiftdistance ←operand2;
result1 ←Overflow32(operand1 << shiftdistance);
```

$R_{NLIDEST}$ ←Register(result1);

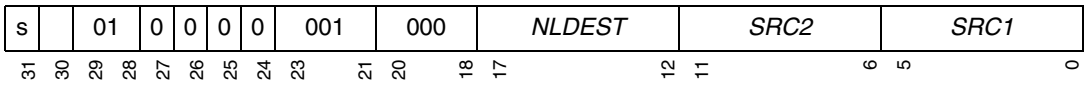
Description: Indicates whether a shls operation with the same input operands would have saturated.

Restrictions: No address or bundle restrictions.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 1 cycle before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

shlso Register

shlso R_{NLDEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
IF (operand2 > 32)
        shiftdistance ←32;
ELSE
        shiftdistance ←operand2;
result1 ←Overflow32(operand1 << shiftdistance);
```

```
RNLDEST ←Register(result1);
```

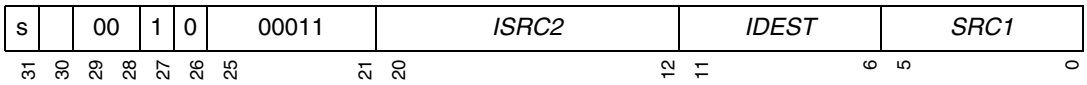
Description: Indicates whether a shls operation with the same input operands would have saturated.

Restrictions: No address or bundle restrictions.
R_{NLDEST} can be any general register except the link register.
There is a latency of 1 cycle before R_{NLDEST} is available for reading.

Exceptions: None.

shr Immediate

shr R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
IF (operand2 > 31)
        result1 ←SignExtend1(Bit(operand1,31));
ELSE
{
        IF (operand2 > 0)
                result1 ←operand1 >> operand2;
        ELSE
                result1 ←operand1;
}
```

R_{IDEST} ←Register(result1);

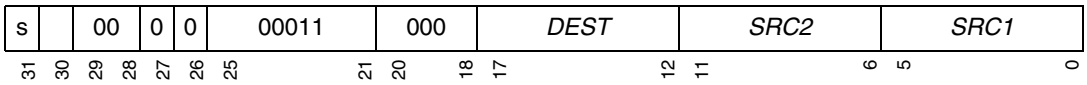
Description: Scalar 32bit arithmetic right shift. Operands may be signed integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shr Register

shr R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
IF (operand2 > 31)
        result1 ←SignExtend1(Bit(operand1,31));
ELSE
{
        IF (operand2 > 0)
                result1 ←operand1 >> operand2;
        ELSE
                result1 ←operand1;
}
```

```
RDEST ←Register(result1);
```

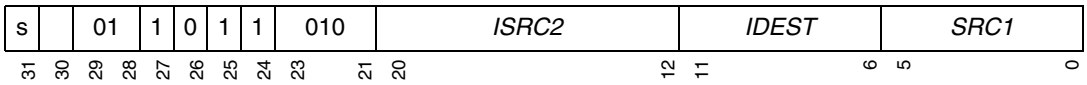
Description: Scalar 32bit arithmetic right shift. Operands may be signed integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shr.ph Immediate

shr.ph R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
result1 ←0;
IF (operand2 > 15)
        shiftdistance ←15;
ELSE
        shiftdistance ←operand2;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←SignedExtract16(operand1, j);
        result1 ←result1 ∨  MaskAndShift16(unpacked_opd1>> shiftdistance, j);
}
```

```
RIDEST ←Register(result1);
```

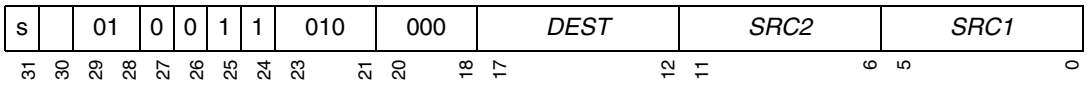
Description: Packed 16bit arithmetic right shift. Operands may be signed 16-bit integer or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shr.ph Register

shr.ph R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
result1 ←0;
IF (operand2 > 15)
        shiftdistance ←15;
ELSE
        shiftdistance ←operand2;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←SignedExtract16(operand1, j);
        result1 ←result1 ∨  MaskAndShift16(unpacked_opd1>> shiftdistance, j);
}
```

```
RDEST ←Register(result1);
```

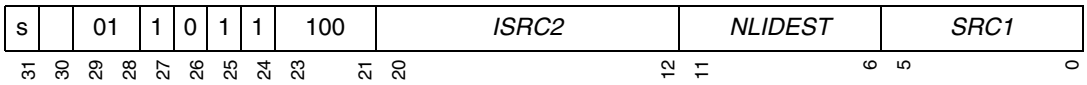
Description: Packed 16bit arithmetic right shift. Operands may be signed 16-bit integer or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shrrne.ph Immediate

shrrne.ph $R_{NLIDEST}$ **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
result1 ←0;
IF (operand2 > 16)
        shiftdistance ←16;
ELSE
        shiftdistance ←operand2;
IF (shiftdistance > 0)
        REPEAT j FROM 0 FOR 2   {
        unpacked_opd1←SignedExtract16(operand1, j);
        rounding ←(1 << (shiftdistance - 1)) + (Bit(unpacked_opd1, shiftdistance) - 1) ;
        result1 ←result1 ∨  MaskAndShift16( (unpacked_opd1+ rounding) >> shiftdistance, j);
}
ELSE
        result1 ←operand1;
```

```
RNLIDEST ←Register(result1);
```

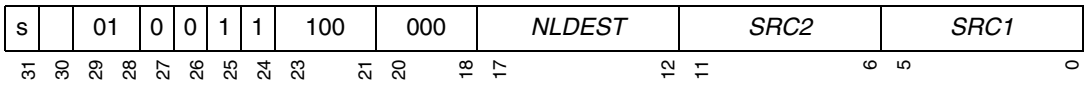
Description: Packed 16-bit signed shift right with round nearest even rounding. Operands may be signed 16-bit integer or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 1 cycle before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

shrrne.ph Register

shrrne.ph R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
result1 ←0;
IF (operand2 > 16)
        shiftdistance ←16;
ELSE
        shiftdistance ←operand2;
IF (shiftdistance > 0)
        REPEAT j FROM 0 FOR 2   {
        unpacked_opd1←SignedExtract16(operand1, j);
        rounding ←(1 << (shiftdistance - 1)) + (Bit(unpacked_opd1, shiftdistance) - 1) ;
        result1 ←result1 ∨  MaskAndShift16( (unpacked_opd1+ rounding) >> shiftdistance, j);
}
ELSE
        result1 ←operand1;
```

```
RNLDEST ←Register(result1);
```

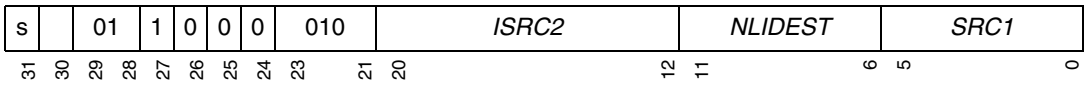
Description: Packed 16-bit signed shift right with round nearest even rounding. Operands may be signed 16-bit integer or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
R_{NLDEST} can be any general register except the link register.
There is a latency of 1 cycle before R_{NLDEST} is available for reading.

Exceptions: None.

shrrnp Immediate

shrrnp $R_{NLIDEST}$ = R_{SRC1}, ISRC2



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
result1 ←0;
IF (operand2 > 32)
        shiftdistance ←32;
ELSE
        shiftdistance ←operand2;
IF (shiftdistance > 0)
{
        rounding ←1 << (shiftdistance - 1);
        result1 ←(operand1 + rounding) >> shiftdistance;
}
ELSE
        result1 ←operand1;
```

```
RNLIDEST ←Register(result1);
```

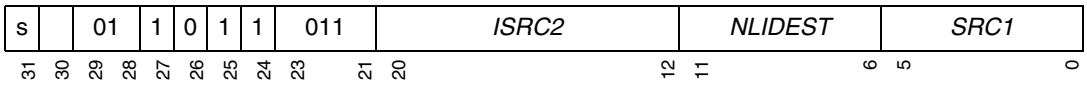
Description: Scalar 32bit arithmetic right shift with round nearest positive rounding. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 1 cycle before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

shrrnp.ph Immediate

shrrnp.ph $R_{NLIDEST}$ **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
result1 ←0;
IF (operand2 > 16)
        shiftdistance ←16;
ELSE
        shiftdistance ←operand2;
IF (shiftdistance > 0)
        REPEAT j FROM 0 FOR 2   {
        unpacked_opd1←SignedExtract16(operand1, j);
        rounding ←1 << (shiftdistance - 1);
        result1 ←result1 ∨  MaskAndShift16( (unpacked_opd1+ rounding) >> shiftdistance, j);
}
ELSE
        result1 ←operand1;
```

```
RNLIDEST ←Register(result1);
```

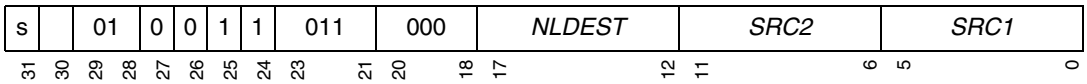
Description: Packed 16-bit signed shift right with round nearest positive rounding. Operands may be signed 16-bit integer or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
$R_{NLIDEST}$ can be any general register except the link register.
There is a latency of 1 cycle before $R_{NLIDEST}$ is available for reading.

Exceptions: None.

shrrnp.ph Register

shrrnp.ph R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
result1 ←0;
IF (operand2 > 16)
        shiftdistance ←16;
ELSE
        shiftdistance ←operand2;
IF (shiftdistance > 0)
        REPEAT j FROM 0 FOR 2   {
        unpacked_opd1←SignedExtract16(operand1, j);
        rounding ←1 << (shiftdistance - 1);
        result1 ←result1 ∨  MaskAndShift16( (unpacked_opd1+ rounding) >> shiftdistance, j);
}
ELSE
        result1 ←operand1;
```

```
RNLDEST ←Register(result1);
```

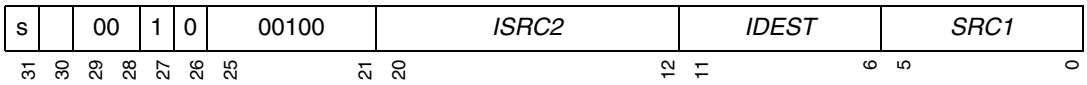
Description: Packed 16-bit signed shift right with round nearest positive rounding. Operands may be signed 16-bit integer or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
R_{NLDEST} can be any general register except the link register.
There is a latency of 1 cycle before R_{NLDEST} is available for reading.

Exceptions: None.

shru Immediate

shru R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
IF (operand2 > 31)
{
        result1 ←0;
}
ELSE
{
        IF (operand2 > 0)
                result1 ←operand1 >> operand2;
        ELSE
                result1 ←operand1;
}
```

R_{IDEST} ←Register(result1);

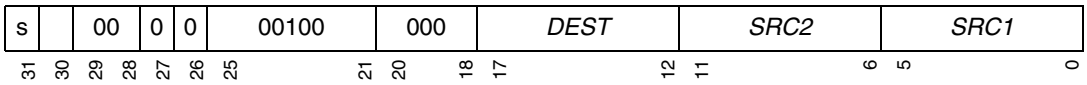
Description: Scalar 32bit logical right shift. Operands are unsigned integers.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shru Register

shru R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
IF (operand2 > 31)
{
        result1 ←0;
}
ELSE
{
        IF (operand2 > 0)
                result1 ←operand1 >> operand2;
        ELSE
                result1 ←operand1;
}
```

```
RDEST ←Register(result1);
```

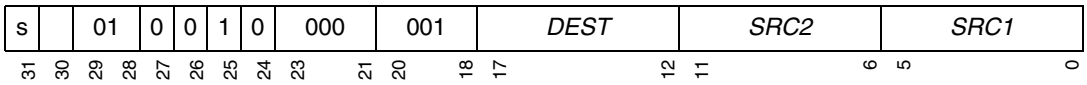
Description: Scalar 32bit logical right shift. Operands are unsigned integers.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shuff.pbh Register

shuff.pbh R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←UnsignedExtract8(operand1, j + 2);
        unpacked_opd2←UnsignedExtract8(operand2, j + 2);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd1, 2 × j);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd2, (2 × j) + 1);
}
```

```
RDEST ←Register(result1);
```

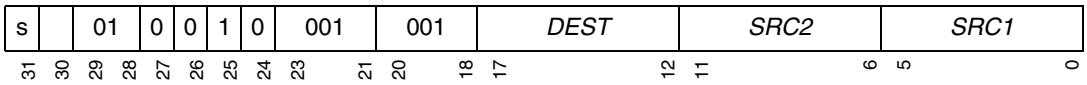
Description: Packed 8-bit shuffle returning high result.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shuff.pbl Register

shuff.pbl R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←UnsignedExtract8(operand1, j);
        unpacked_opd2←UnsignedExtract8(operand2, j);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd1, 2 × j);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd2, (2 × j) + 1);
}
```

R_{DEST} ←Register(result1);

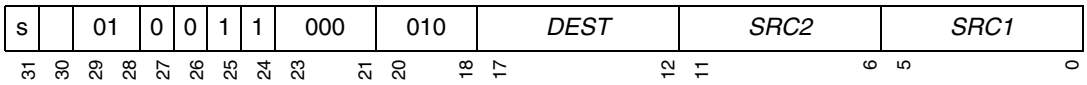
Description: Packed 8-bit shuffle returning low result.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shuff.phh Register

shuff.phh R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
unpacked_opd1←UnsignedExtract$_{16}$(operand1, 1);
unpacked_opd2←UnsignedExtract$_{16}$(operand2, 1);
result1 ←unpacked_opd1∨ MaskAndShift$_{16}$(unpacked_opd2, 1);

R_{DEST} ←Register(result1);

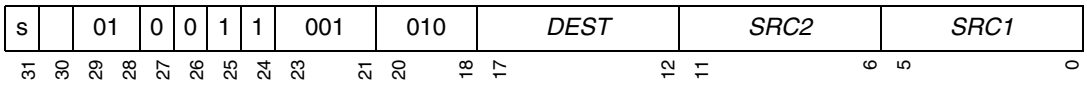
Description: Packed 16-bit shuffle returning high result.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shuff.phl Register

shuff.phl R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});
unpacked_opd1←UnsignedExtract$_{16}$(operand1, 0);
unpacked_opd2←UnsignedExtract$_{16}$(operand2, 0);
result1 ←unpacked_opd1∨ MaskAndShift$_{16}$(unpacked_opd2, 1);

R_{DEST} ←Register(result1);

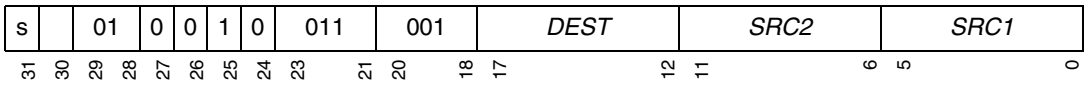
Description: Packed 16-bit shuffle returning low result.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shuffeve.pb Register

shuffeve.pb R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←UnsignedExtract8(operand1, 2 × j);
        unpacked_opd2←UnsignedExtract8(operand2, 2 × j);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd1, 2 × j);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd2, (2 × j) + 1);
}
```

```
RDEST ←Register(result1);
```

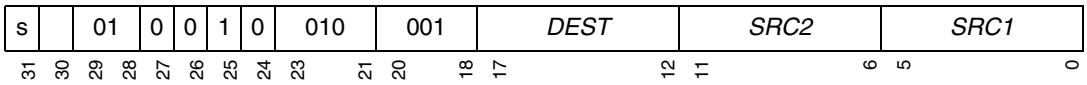
Description: Packed 8-bit shuffle of even fields.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

shuffodd.pb Register

shuffodd.pb R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        unpacked_opd1←UnsignedExtract8(operand1, (2 × j) + 1);
        unpacked_opd2←UnsignedExtract8(operand2, (2 × j) + 1);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd1, 2 × j);
        result1 ←result1 ∨  MaskAndShift8(unpacked_opd2, (2 × j) + 1);
}
```

```
RDEST ←Register(result1);
```

Description: Packed 8-bit shuffle of odd fields.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

slct Immediate

slct R_{IDEST} = B_{SCOND}, R_{SRC1}, ISRC2

Semantics:

```
operand1 ←ZeroExtend4(BSCOND);
operand2 ←SignExtend32(RSRC1);
operand3 ←SignExtend32(Imm(ISRC2));
IF (operand1 ≠ 0)
        result1 ←operand2;
ELSE
        result1 ←operand3;
```

```
RIDEST ←Register(result1);
```

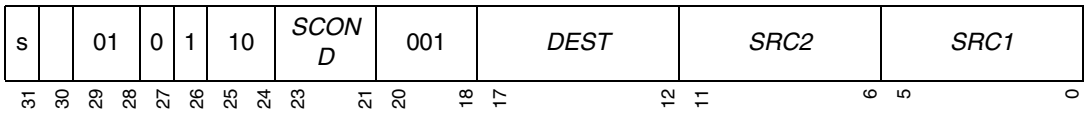
Description: Conditional select using logical value of branch register.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

slct Register

slct R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend4(BSCOND);
operand2 ←SignExtend32(RSRC1);
operand3 ←SignExtend32(RSRC2);
IF (operand1 = 0)
        result1 ←operand3;
ELSE
        result1 ←operand2;
```

R_{DEST} ←Register(result1);

Description: Conditional select using logical value of branch register.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

slct.pb Immediate

slct.pb R_{IDEST} = B_{SCOND}, R_{SRC1}, ISRC2

Semantics:

```
operand1 ←ZeroExtend4(BSCOND);
operand2 ←ZeroExtend32(RSRC1);
operand3 ←ZeroExtend32(Imm(ISRC2));
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        IF (UnsignedExtract1(operand1, j) = 0)
        {
                byte ←UnsignedExtract8(operand3, j);
                result1 ←result1 ∨  MaskAndShift8(byte, j);
        }
        ELSE
        {
                byte ←UnsignedExtract8(operand2, j);
                result1 ←result1 ∨  MaskAndShift8(byte, j);
        }
}
```

R_{IDEST} ←Register(result1);

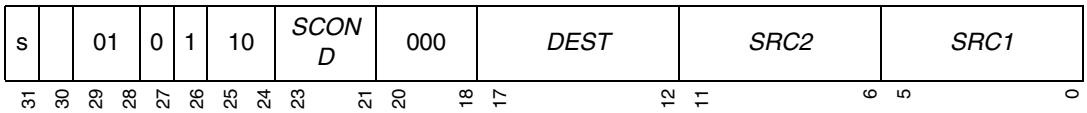
Description: Byte select between input operands using 4-bit condition code in branch register.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

slct.pb Register

slct.pb R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend4(BSCOND);
operand2 ←ZeroExtend32(RSRC1);
operand3 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        IF (UnsignedExtract1(operand1, j) = 0)
        {
                byte ←UnsignedExtract8(operand3, j);
                result1 ←result1 ∨  MaskAndShift8(byte, j);
        }
        ELSE
        {
                byte ←UnsignedExtract8(operand2, j);
                result1 ←result1 ∨  MaskAndShift8(byte, j);
        }
}
```

R_{DEST} ←Register(result1);

Description: Byte select between input operands using 4-bit condition code in branch register.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

slctf Immediate

slctf R_{IDEST} = B_{SCOND}, R_{SRC1}, ISRC2

Semantics:

```
operand1 ←ZeroExtend4(BSCOND);
operand2 ←SignExtend32(RSRC1);
operand3 ←SignExtend32(Imm(ISRC2));
IF (operand1 = 0)
        result1 ←operand2;
ELSE
        result1 ←operand3;
```

```
RIDEST ←Register(result1);
```

Description: Conditional select using negated logical value of branch register.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

slctf.pb Immediate

slctf.pb R_{IDEST} = B_{SCOND}, R_{SRC1}, ISRC2

Semantics:

```
operand1 ←ZeroExtend4(BSCOND);
operand2 ←ZeroExtend32(RSRC1);
operand3 ←ZeroExtend32(Imm(ISRC2));
result1 ←0;
REPEAT j FROM 0 FOR 4 {
        IF (UnsignedExtract1(operand1, j) = 0)
        {
                byte ←UnsignedExtract8(operand2, j);
                result1 ←result1 ∨  MaskAndShift8(byte, j);
        }
        ELSE
        {
                byte ←UnsignedExtract8(operand3, j);
                result1 ←result1 ∨  MaskAndShift8(byte, j);
        }
}
```

R_{IDEST} ←Register(result1);

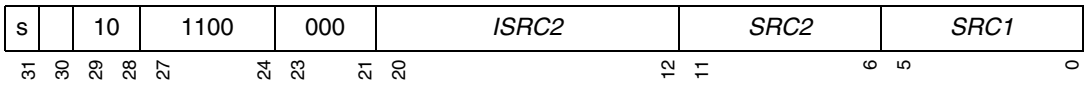
Description: Byte select between input operands using negated 4-bit condition code in branch register.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

stb

stb ISRC2[R_{SRC1}] = R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
operand3 ←SignExtend32(RSRC2);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakStoreHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOLATION;
WriteCheckMemory8(ea);
```

```
WriteMemory8(ea, operand3);
```

Description: Store byte.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB

stbc

stbc ISRC2[R_{SRC1}] = B_{PCOND}, R_{SRC2}

Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
operand3 ←ZeroExtend4(BPCOND);
operand4 ←SignExtend32(RSRC2);
IF (operand3 ≠ 0)
{
        ea ←ZeroExtend32(operand1 + operand2);
        IF (IsDBreakStoreHit(ea))
                THROW DBREAK;
        IF (IsCRegSpace(ea))
                THROW CREG_ACCESS_VIOLATION;
        WriteCheckMemory8(ea);
}
```

```
IF (operand3 ≠ 0)
        WriteMemory8(ea, operand4);
```

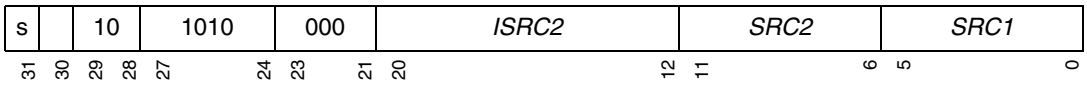
Description: Conditional store byte.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
B_{PCOND} can be any branch register except zero.
No latency constraints.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB

sth

sth ISRC2[R_{SRC1}] = R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
operand3 ←SignExtend32(RSRC2);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakStoreHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOLATION;
WriteCheckMemory16(ea);
```

```
WriteMemory16(ea, operand3);
```

Description: Store half-word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

sthc

sthc ISRC2[R_{SRC1}] = B_{PCOND}, R_{SRC2}

Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
operand3 ←ZeroExtend4(BPCOND);
operand4 ←SignExtend32(RSRC2);
IF (operand3 ≠ 0)
{
        ea ←ZeroExtend32(operand1 + operand2);
        IF (IsDBreakStoreHit(ea))
                THROW DBREAK;
        IF (IsCRegSpace(ea))
                THROW CREG_ACCESS_VIOLATION;
        WriteCheckMemory16(ea);
}
```

```
IF (operand3 ≠ 0)
        WriteMemory16(ea, operand4);
```

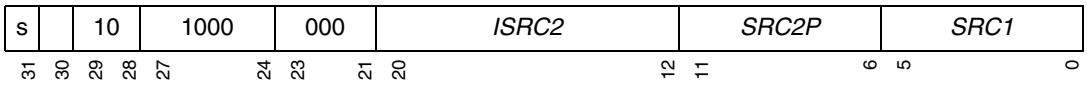
Description: Conditional store half-word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
B_{PCOND} can be any branch register except zero.
No latency constraints.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

stl

stl ISRC2[R_{SRC1}] = P_{SRC2P}



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
operand3 ←0;
IF (SRC2P ≠ 0)
{
        operand3 ←ZeroExtend64(RSRC2P);
        operand3 ←operand3 ∨  (ZeroExtend64(R(SRC2P + 1)) <<32);
}
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakStoreHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOLATION;
WriteCheckMemory64(ea);
```

```
WriteMemory64(ea, operand3);
```

Description: Store double word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
P_{SRC2P} can be any register pair.
No latency constraints.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

stlc

stlc ISRC2[R_{SRC1}] = B_{PCOND}, P_{SRC2P}

Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(R_SRC1);
operand3 ←ZeroExtend4(B_PCOND);
operand4 ←0;
IF (SRC2P ≠ 0)
{
        operand4 ←ZeroExtend64(R_SRC2P);
        operand4 ←operand4 ∨  (ZeroExtend64(R_(SRC2P + 1)) <<32);
}
IF (operand3 ≠ 0)
{
        ea ←ZeroExtend32(operand1 + operand2);
        IF (IsDBreakStoreHit(ea))
                THROW DBREAK;
        IF (IsCRegSpace(ea))
                THROW CREG_ACCESS_VIOLATION;
        WriteCheckMemory64(ea);
}
```

```
IF (operand3 ≠ 0)
        WriteMemory64(ea, operand4);
```

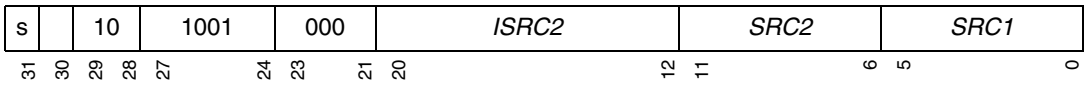
Description: Conditional store double word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
B_{PCOND} can be any branch register except zero.
P_{SRC2P} can be any register pair.
No latency constraints.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

stw

stw ISRC2[R_{SRC1}] = R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
operand3 ←SignExtend32(RSRC2);
ea ←ZeroExtend32(operand1 + operand2);
IF (IsDBreakStoreHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        WriteCheckCReg(ea);
ELSE
        WriteCheckMemory32(ea);
```

```
IF (IsCRegSpace(ea))
        WriteCReg(ea, operand3);
ELSE
        WriteMemory32(ea, operand3);
```

Description: Store word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: DBREAK, DTLB, CREG_ACCESS_VIOLATION, CREG_NO_MAPPING, MISALIGNED_TRAP

stwc

stwc ISRC2[R_{SRC1}] = B_{PCOND}, R_{SRC2}

Semantics:

```
operand1 ←SignExtend32(Imm(ISRC2));
operand2 ←SignExtend32(RSRC1);
operand3 ←ZeroExtend4(BPCOND);
operand4 ←SignExtend32(RSRC2);
IF (operand3 ≠ 0)
{
        ea ←ZeroExtend32(operand1 + operand2);
        IF (IsDBreakStoreHit(ea))
                THROW DBREAK;
        IF (IsCRegSpace(ea))
                WriteCheckCReg(ea);
        ELSE
                WriteCheckMemory32(ea);
}
```

```
IF (operand3 ≠ 0)
{
        IF (IsCRegSpace(ea))
                WriteCReg(ea, operand4);
        ELSE
                WriteMemory32(ea, operand4);
}
```

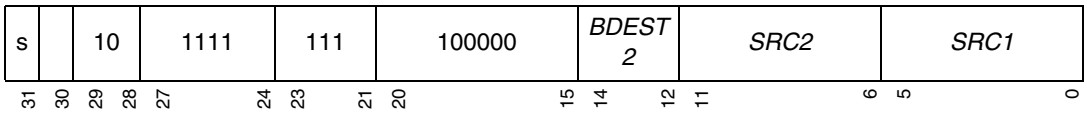
Description: Conditional store word.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
B_{PCOND} can be any branch register except zero.
No latency constraints.

Exceptions: DBREAK, DTLB, CREG_ACCESS_VIOLATION, CREG_NO_MAPPING, MISALIGNED_TRAP

stwl

stwl B_{BDEST2}, [R_{SRC1}] = R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
storeCondition ←ATOMIC_LOCK[LOCKED];
IF (storeCondition)
        result1 ←1;
ELSE
        result1 ←0;
ATOMIC_LOCK←0;
IF (IsDBreakStoreHit(operand1))
        THROW DBREAK;
IF (IsCRegSpace(operand1))
        THROW CREG_ACCESS_VIOLATION;
WriteCheckMemory32(operand1);
```

```
IF (storeCondition)
        WriteMemory32(operand1, operand2);
BBDEST2 ←BranchRegister(result1);
```

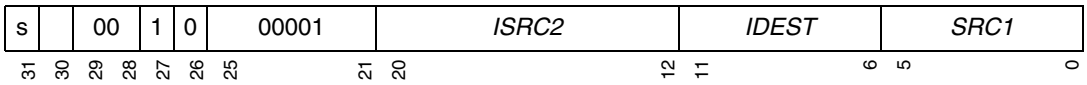
Description: Store word conditional linked. Forms part of atomic read/modify/write sequence with ldwl.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
There is a latency of 2 cycles before B_{BDEST2} is available for reading by br/brf.
No other latency restrictions before B_{BDEST2} is available for reading by any other operation.

Exceptions: DBREAK, CREG_ACCESS_VIOLATION, DTLB, MISALIGNED_TRAP

sub Immediate

sub R_{IDEST} **=** ISRC2**,** R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand2 - operand1;

R_{IDEST} ←Register(result1);

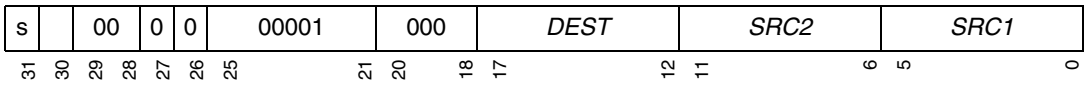
Description: Scalar 32bit subtraction. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sub Register

sub R_{DEST} = R_{SRC2}, R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand2 - operand1;

R_{DEST} ←Register(result1);

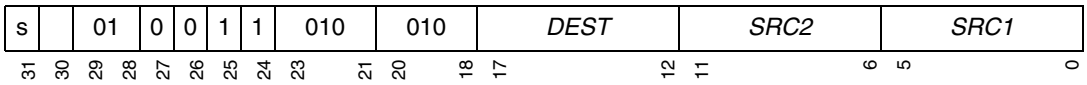
Description: Scalar 32bit subtraction. Operands may be signed or unsigned integers or fractional 1.31 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sub.ph Register

sub.ph R_{DEST} **=** R_{SRC2}**,** R_{SRC1}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        subresult ←SignedExtract16(operand2, j) - SignedExtract16(operand1, j);
        result1 ←result1 ∨  MaskAndShift16(subresult, j);
}
```

```
RDEST ←Register(result1);
```

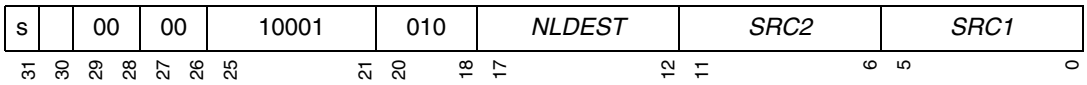
Description: Packed 16-bit subtraction. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

subf.n Floating point - Register

subf.n R_{NLDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{32}$(R_{SRC2});

result1 ←FSubSNonIeee(operand1, operand2);
R_{NLDEST} ←Register(result1);

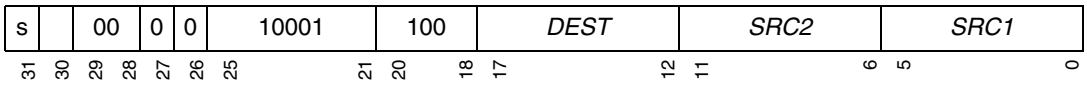
Description: IEEE754 format single precision floating point subtraction.

Restrictions: Must be encoded at an even word address.
R_{NLDEST} can be any general register except the link register.
There is a latency of 2 cycles before R_{NLDEST} is available for reading.

Exceptions: None.

subs Register

subs R_{DEST} = R_{SRC2}, R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←Saturate$_{32}$(operand2 - operand1);

R_{DEST} ←Register(result1);

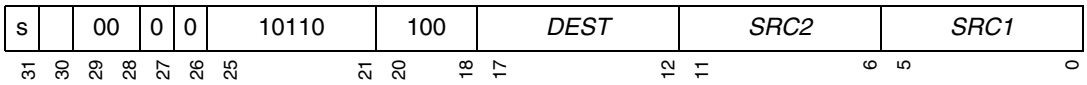
Description: Scalar 32bit subtraction with saturation. Operands may be signed integers or fractional 1.31 values.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

subs.ph Register

subs.ph R_{DEST} **=** R_{SRC2}**,** R_{SRC1}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
result1 ←0;
REPEAT j FROM 0 FOR 2 {
        subresult ←SignedExtract16(operand2, j) - SignedExtract16(operand1, j);
        subresult ←Saturate16(subresult);
        result1 ←result1 ∨  MaskAndShift16(subresult, j);
}
```

```
RDEST ←Register(result1);
```

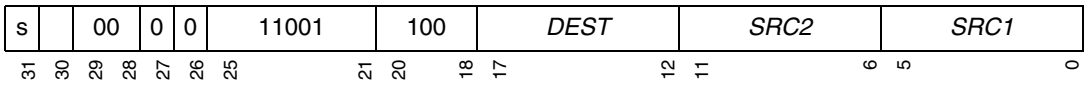
Description: Packed 16-bit signed subtraction with saturation. Operands may be signed or unsigned 16-bit integers or fractional 1.15 format.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

subso Register

subso R_{DEST} = R_{SRC2}, R_{SRC1}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←Overflow$_{32}$(operand2 - operand1);

R_{DEST} ←Register(result1);

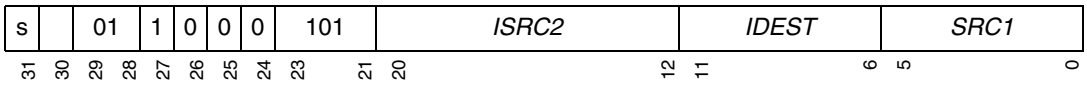
Description: Indicates whether a subs operation with the given input operands causes a saturation.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sxt Immediate

sxt R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
IF (operand2 = 0)
        result1 ←0;
ELSE
{
        IF (operand2 > 31)
        {
                result1 ←operand1;
        }
        ELSE
        {
                sign ←(1 << (operand2 - 1));
                mask ←(1 << (operand2 - 1)) - 1;
                IF (operand1 ∧ sign)
                        result1 ←operand1 ∨  (~mask);
                ELSE
                        result1 ←operand1 ∧ mask;
        }
}
```

```
RIDEST ←Register(result1);
```

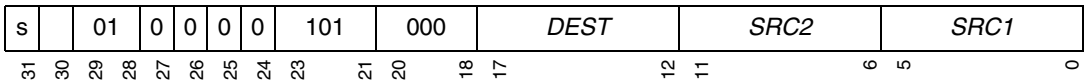
Description: Arbitrary sign extend.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sxt Register

sxt R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
IF (operand2 = 0)
        result1 ←0;
ELSE
{
        IF (operand2 > 31)
        {
                result1 ←operand1;
        }
        ELSE
        {
                sign ←(1 << (operand2 - 1));
                mask ←(1 << (operand2 - 1)) - 1;
                IF (operand1 ∧ sign)
                        result1 ←operand1 ∨  (~mask);
                ELSE
                        result1 ←operand1 ∧ mask;
        }
}
```

R_{DEST} ←Register(result1);

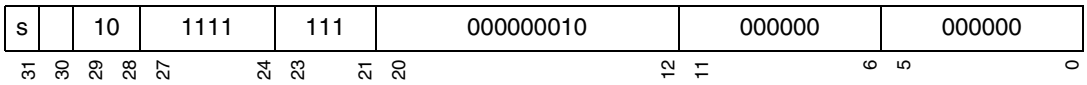
Description: Arbitrary sign extend.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

sync

sync



Semantics:

```
Sync();
```

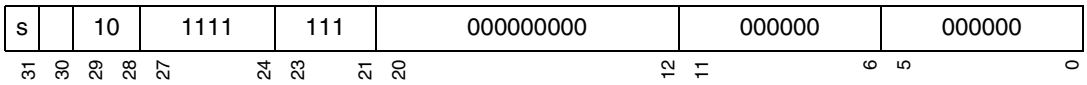
Description: Ensure that all outstanding memory transactions have completed.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: None.

syncins

syncins



Semantics:

```
SyncIns();
```

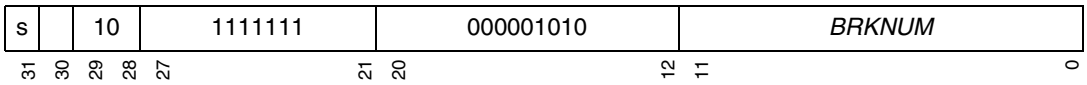
Description: Ensure that all previous operations have completed and no new operations have started.

Restrictions: Must be the first in a bundle and uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: None.

syscall

syscall BRKNUM



Semantics:

operand1 ←ZeroExtend$_{12}$(BRKNUM);
THROW SYSCALL;

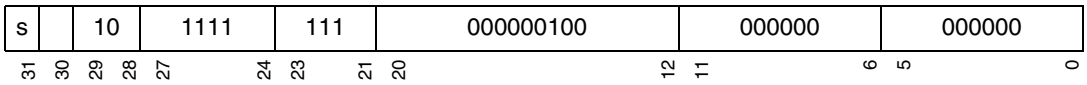
Description: System call.

Restrictions: Must be the only operation in a bundle.
No latency constraints.

Exceptions: SYSCALL

waitl

waitl



Semantics:

```
WaitForLink();
```

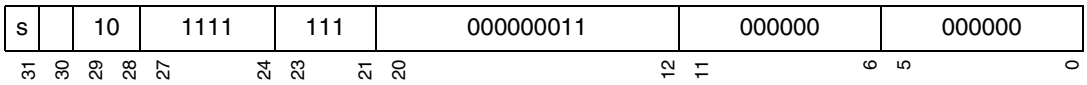
Description: Wait for link. Used to make an atomic read/modify/write sequence for a multi-processor or multi-threaded implementation more efficient in conjunction with ldwl/stwl. Waitl is executed as a nop by a uniprocessor.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: None.

wmb

wmb



Semantics:

```
Wmb();
```

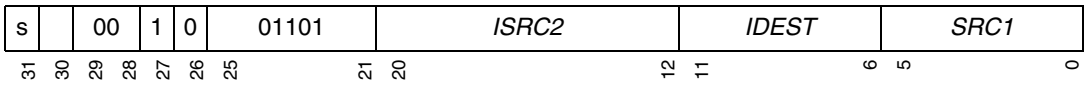
Description: Broadcast all pending memory writes in a multi-processor system and wait for responses. This is executed as a nop in a uniprocessor or a multi-threaded implementation.

Restrictions: Uses the ld/st unit for which only one operation is allowed per bundle.
No latency constraints.

Exceptions: None.

xor Immediate

xor R_{IDEST} = R_{SRC1}, ISRC2



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(ISRC2));
result1 ←operand1 ⊕ operand2;

R_{IDEST} ←Register(result1);

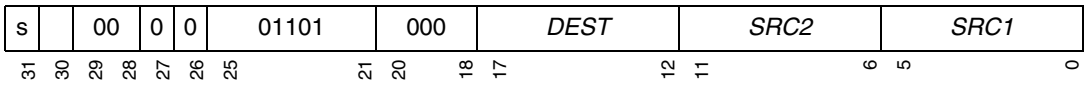
Description: Bitwise XOR.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

xor Register

xor R_{DEST} **=** R_{SRC1}**,** R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});
result1 ←operand1 ⊕ operand2;

R_{DEST} ←Register(result1);

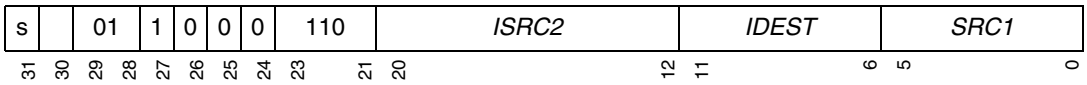
Description: Bitwise XOR.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

zxt Immediate

zxt R_{IDEST} **=** R_{SRC1}**,** ISRC2



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(Imm(ISRC2));
IF (operand2 > 31)
{
        result1 ←operand1;
}
ELSE
{
        mask ←(1 << operand2) - 1;
        result1 ←operand1 ∧ mask;
}
```

R_{IDEST} ←Register(result1);

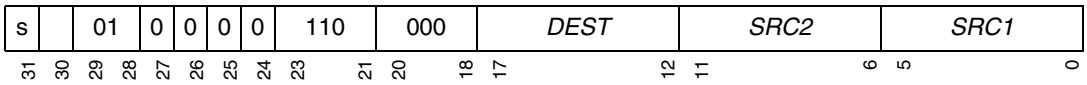
Description: Arbitrary zero extend.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

zxt Register

zxt R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend8(RSRC2);
IF (operand2 > 31)
{
        result1 ←operand1;
}
ELSE
{
        mask ←(1 << operand2) - 1;
        result1 ←operand1 ∧ mask;
}
```

```
RDEST ←Register(result1);
```

Description: Arbitrary zero extend.

Restrictions: No address or bundle restrictions.
No latency constraints.

Exceptions: None.

Appendix A Instruction encoding

This appendix provides a description of the ST240 instruction encoding.

A.1 Reserved bits

Any bits that are not defined are reserved. These bits must be set to 0.

A.2 Fields

Each instruction encoding is composed of a number of fields representing the operands. These are detailed in *Table 162*.

Table 162. Operand fields

Operand field	Description
BCOND	Branch register containing the branch condition.
BDEST	Destination branch register for register format operations.
BDEST2	Destination branch register.
BSRC1	Branch register as first source operand
BSRC2	Branch register as second source operand
BTARG	Branch offset value from PC.
DEST	Destination general purpose register for register format operations.
IBDEST	Destination branch register for immediate format operations.
IDEST	Destination general purpose register for immediate format operations.
IDESTP	Destination general purpose register pair for immediate format operations.
ISRC2	9-bit short immediate value.
IMM	23-bit value used to extend a short immediate.
NLDEST	Destination general purpose register for multiply operations ($r63 cannot be used).
NLIDEST	Destination general purpose register for immediate format multiplies ($r63 cannot be used).
SCOND	Source branch register used for select condition or carry.
PCOND	Source branch register used for conditional ld/st operations ($b0 cannot be used).
SRC1	General purpose source register.
SRC2	General purpose source register.
SRC2P	General purpose source register pair.
BRKNUM	12-bit immediate operand for sbrk

Table 163. Formats

	Stop bit		Format	Opcode							Opcode2/ Immediate/ Dest								Dest/Src2						Src1					
	31	30	29 28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
SIMDCMPH_R	s		00	0	1	0	1	OPC			001			*DEST*					*SRC2*						*SRC1*					
SIMDCMPB_R	s		00	0	1	0	0	OPC			001			*DEST*					*SRC2*						*SRC1*					
SIMDCMPH_B	s		00	0	1	1	1	OPC			001000						*BDEST2*		*SRC2*						*SRC1*					
SIMDCMPB_B	s		00	0	1	1	0	OPC			001000						*BDEST2*		*SRC2*						*SRC1*					
SIMDH	s		01	0	0	1	1	OPC			001			*DEST*					*SRC2*						*SRC1*					
SIMDH2	s		01	0	0	1	1	OPC			010			*DEST*					*SRC2*						*SRC1*					
SIMDB	s		01	0	0	1	0	OPC			001			*DEST*					*SRC2*						*SRC1*					
SIMDB2	s		01	0	0	1	0	OPC			010			*DEST*					*SRC2*						*SRC1*					
SIMDHRI_R	s		01	0	0	1	1	OPC			000			*DEST*					*SRC2*						*SRC1*					
SIMDHRI_I	s		01	1	0	1	1	OPC			*ISRC2*								*IDEST*						*SRC1*					
SIMDBRI_R	s		01	0	0	1	0	OPC			000			*DEST*					*SRC2*						*SRC1*					
SIMDBRI_I	s		01	1	0	1	0	OPC			*ISRC2*								*IDEST*						*SRC1*					
ARITH_R	s		01	0	0	0	0	OPC			000			*DEST*					*SRC2*						*SRC1*					
ARITH_I	s		01	1	0	0	0	OPC			*ISRC2*								*IDEST*						*SRC1*					
ARITH_I2	s		01	1	0	0	1	OPC			*ISRC2*								*IDEST*						*SRC1*					
Br3R	s		00	0	1	1	OPC				000001						*BDEST2*				*BSRC2*						*BSRC1*			
Br2R	s		00	0	1	1	OPC				000010						*BDEST2*										*BSRC1*			
Int3R	s		00	0	0	OPC					000			*DEST*					*SRC2*						*SRC1*					
Int3I	s		00	1	0	OPC					*ISRC2*								*IDEST*						*SRC1*					
Monadic	s		00	1	0	01110					OPC								*IDEST*						*SRC1*					
Cmp3R_Reg	s		00	0	1	0	OPC				000			*DEST*					*SRC2*						*SRC1*					
Cmp3R_Br	s		00	0	1	1	OPC				000000						*BDEST2*		*SRC2*						*SRC1*					
Cmp3I_Reg	s		00	1	1	0	OPC				*ISRC2*								*IDEST*						*SRC1*					
Cmp3I_Br	s		00	1	1	1	OPC				*ISRC2*										*IBDEST*				*SRC1*					
Imm	s		01	0	1	01	OPC				*IMM*																			
SelectR	s		01	0	1	OPC	*SCOND*				OPC			*DEST*					*SRC2*						*SRC1*					
SelectI	s		01	1	1	OPC	*SCOND*				*ISRC2*								*IDEST*						*SRC1*					
cgen	s		01	0	1	OPC	*SCOND*				*BDEST*			*DEST*					*SRC2*						*SRC1*					
Break	s		10	1111111							OPC								*BRKNUM*											
Load	s		10	OPC			000				*ISRC2*								*NLIDEST*						*SRC1*					
LoadC	s		10	OPC			*PCOND*				*ISRC2*								*NLIDEST*						*SRC1*					
LoadL	s		10	1111			111				OPC								*IDEST*						*SRC1*					
MemSub	s		10	OPC			000				*ISRC2*								000000						*SRC1*					
Psw	s		10	OPC			LS_SUBOPCODE				*ISRC2*								*NLIDEST*						*SRC1*					
Store	s		10	OPC			000				*ISRC2*								*SRC2*						*SRC1*					
StoreC	s		10	OPC			*PCOND*				*ISRC2*								*SRC2*						*SRC1*					
StoreL	s		10	OPC			111				OPC						*BDEST2*		*SRC2*						*SRC1*					
System	s		10	1111			111				OPC								*SRC2*						*SRC1*					

Table 163. Formats (Continued)

	Stop bit		Format		Opcode							Opcode2/ Immediate/ Dest								Dest/Src2						Src1					
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Call	s		11		0	OPC			LNK	*BTARG*																						
Branch	s		11		1	OPC	*BCOND*			*BTARG*																						
Maths3R_Dss	s		00		00		OPC					010	*NLDEST*								*SRC2*						*SRC1*					
Maths3R_Mss	s		00		00		OPC					010	*NLDEST*														*SRC1*					

Important points to note.

- The **stop** bit indicates the end of bundle and is set in the last syllable of the bundle.
- The format bits are used to decode the class of operation. There are four formats:

  Integer — arithmetic, comparison

  Specific — immediate extension, selects, extended arithmetic

  Memory — load, store

  Control transfer — branch, call, **rfi**, **goto**

- Additional decoding is performed using the most significant instruction bits.
- **Int3** operations have two base formats, register (Int3R) and immediate (Int3I). Bit 27 specifies the Int3 format, 0 = register format, 1 = immediate format. In register format, the operation consists of R_{DEST} = R_{SRC1} Op RSRC2. Immediate format consists of R_{DEST} = R_{SRC1} Op IMMEDIATE.
- **Cmp3** format is similar to Int3 except it can have as a destination either a general purpose register or a branch register (BBDEST). In register format, the target register specifier occupies bits 12 to 17, while the target branch register bits 18 to 20. In immediate format, bits 6 to 11 specify either the target general purpose register or target branch register (bits 6 to 8).
- **Load** operations follow R_{DEST} = Mem[R_{SRC1} + IMMEDIATE] semantics, while stores follow Mem[R_{SRC1} + IMMEDIATE] = R_{SRC2}. Thus bits 6 to 11 specify either the target destination register (R_{DEST}) or the second operand source register (R_{SRC2}), depending on whether the operation is a load or store.

A.3 Opcodes

Table 164. Instruction encoding

	31	30	29 28	27 26	25 24 23 22 21	20 19 18	17 16 15 14 13 12	11 10 9 8 7 6	5 4 3 2 1 0
add	s		00	0 0	00000	000	*DEST*	*SRC2*	*SRC1*
sub	s		00	0 0	00001	000	*DEST*	*SRC2*	*SRC1*
shl	s		00	0 0	00010	000	*DEST*	*SRC2*	*SRC1*
shr	s		00	0 0	00011	000	*DEST*	*SRC2*	*SRC1*
shru	s		00	0 0	00100	000	*DEST*	*SRC2*	*SRC1*
sh1add	s		00	0 0	00101	000	*DEST*	*SRC2*	*SRC1*
sh2add	s		00	0 0	00110	000	*DEST*	*SRC2*	*SRC1*
sh3add	s		00	0 0	00111	000	*DEST*	*SRC2*	*SRC1*
and	s		00	0 0	01001	000	*DEST*	*SRC2*	*SRC1*
andc	s		00	0 0	01010	000	*DEST*	*SRC2*	*SRC1*
or	s		00	0 0	01011	000	*DEST*	*SRC2*	*SRC1*
orc	s		00	0 0	01100	000	*DEST*	*SRC2*	*SRC1*
xor	s		00	0 0	01101	000	*DEST*	*SRC2*	*SRC1*
mul64h	s		00	0 0	01111	000	*NLDEST*	*SRC2*	*SRC1*
max	s		00	0 0	10000	000	*DEST*	*SRC2*	*SRC1*
addf.n	s		00	00	10000	010	*NLDEST*	*SRC2*	*SRC1*
adds	s		00	0 0	10000	100	*DEST*	*SRC2*	*SRC1*
maxu	s		00	0 0	10001	000	*DEST*	*SRC2*	*SRC1*
subf.n	s		00	00	10001	010	*NLDEST*	*SRC2*	*SRC1*
subs	s		00	0 0	10001	100	*DEST*	*SRC2*	*SRC1*
min	s		00	0 0	10010	000	*DEST*	*SRC2*	*SRC1*
mulf.n	s		00	00	10010	010	*NLDEST*	*SRC2*	*SRC1*
sh1adds	s		00	0 0	10010	100	*DEST*	*SRC2*	*SRC1*
minu	s		00	0 0	10011	000	*DEST*	*SRC2*	*SRC1*
convif.n	s		00	00	10011	010	*NLDEST*		*SRC1*
sh1subs	s		00	0 0	10011	100	*DEST*	*SRC2*	*SRC1*
mul64hu	s		00	0 0	10100	000	*NLDEST*	*SRC2*	*SRC1*
convfi.n	s		00	00	10100	010	*NLDEST*		*SRC1*
sats	s		00	0 0	10100	100	*DEST*	000000	*SRC1*
mull	s		00	0 0	10101	000	*NLDEST*	*SRC2*	*SRC1*
adds.ph	s		00	0 0	10101	100	*DEST*	*SRC2*	*SRC1*
mul32	s		00	0 0	10110	000	*NLDEST*	*SRC2*	*SRC1*
subs.ph	s		00	0 0	10110	100	*DEST*	*SRC2*	*SRC1*
mulh	s		00	0 0	10111	000	*NLDEST*	*SRC2*	*SRC1*
div	s		00	00	11000	010	*NLDEST*	*SRC2*	*SRC1*
addso	s		00	0 0	11000	100	*DEST*	*SRC2*	*SRC1*
mulll	s		00	0 0	11001	000	*NLDEST*	*SRC2*	*SRC1*
rem	s		00	00	11001	010	*NLDEST*	*SRC2*	*SRC1*
subso	s		00	0 0	11001	100	*DEST*	*SRC2*	*SRC1*
mulllu	s		00	0 0	11010	000	*NLDEST*	*SRC2*	*SRC1*
divu	s		00	00	11010	010	*NLDEST*	*SRC2*	*SRC1*
sh1addso	s		00	0 0	11010	100	*DEST*	*SRC2*	*SRC1*
mullh	s		00	0 0	11011	000	*NLDEST*	*SRC2*	*SRC1*
remu	s		00	00	11011	010	*NLDEST*	*SRC2*	*SRC1*

Table 164. Instruction encoding (Continued)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sh1subso	s		00		0	0	11011					100			*DEST*						*SRC2*						*SRC1*					
mullhu	s		00		0	0	11100					000			*NLDEST*						*SRC2*						*SRC1*					
satso	s		00		0	0	11100					100			*DEST*						000000						*SRC1*					
mulhh	s		00		0	0	11101					000			*NLDEST*						*SRC2*						*SRC1*					
mulhhu	s		00		0	0	11110					000			*NLDEST*						*SRC2*						*SRC1*					
mulfrac	s		00		0	0	11111					000			*NLDEST*						*SRC2*						*SRC1*					
cmpeq	s		00		0	1	0	0000				000			*DEST*						*SRC2*						*SRC1*					
cmpeq.pb	s		00		0	1	0	0	000			001			*DEST*						*SRC2*						*SRC1*					
cmpne	s		00		0	1	0	0001				000			*DEST*						*SRC2*						*SRC1*					
cmpgef.n	s		00		0	1	0	0010				000			*DEST*						*SRC2*						*SRC1*					
cmpgtu.pb	s		00		0	1	0	0	010			001			*DEST*						*SRC2*						*SRC1*					
cmpeqf.n	s		00		0	1	0	0011				000			*DEST*						*SRC2*						*SRC1*					
cmpgtf.n	s		00		0	1	0	0100				000			*DEST*						*SRC2*						*SRC1*					
cmple	s		00		0	1	0	0110				000			*DEST*						*SRC2*						*SRC1*					
cmpleu	s		00		0	1	0	0111				000			*DEST*						*SRC2*						*SRC1*					
cmplt	s		00		0	1	0	1000				000			*DEST*						*SRC2*						*SRC1*					
cmpeq.ph	s		00		0	1	0	1	000			001			*DEST*						*SRC2*						*SRC1*					
cmpltu	s		00		0	1	0	1001				000			*DEST*						*SRC2*						*SRC1*					
cmpgt.ph	s		00		0	1	0	1	001			001			*DEST*						*SRC2*						*SRC1*					
andl	s		00		0	1	0	1010				000			*DEST*						*SRC2*						*SRC1*					
nandl	s		00		0	1	0	1011				000			*DEST*						*SRC2*						*SRC1*					
orl	s		00		0	1	0	1100				000			*DEST*						*SRC2*						*SRC1*					
norl	s		00		0	1	0	1101				000			*DEST*						*SRC2*						*SRC1*					
cmpeq	s		00		0	1	1	0000				000000						*BDEST 2*			*SRC2*						*SRC1*					
andl	s		00		0	1	1	0000				000001						*BDEST 2*					*BSRC2*						*BSRC1*			
mov	s		00		0	1	1	0000				000010						*BDEST 2*												*BSRC1*		
cmpeq.pb	s		00		0	1	1	0	000			001000						*BDEST 2*			*SRC2*						*SRC1*					
cmpne	s		00		0	1	1	0001				000000						*BDEST 2*			*SRC2*						*SRC1*					
nandl	s		00		0	1	1	0001				000001						*BDEST 2*					*BSRC2*						*BSRC1*			
cmpgtu.pb	s		00		0	1	1	0	001			001000						*BDEST 2*			*SRC2*						*SRC1*					
cmpgef.n	s		00		0	1	1	0010				000000						*BDEST 2*			*SRC2*						*SRC1*					
orl	s		00		0	1	1	0010				000001						*BDEST 2*					*BSRC2*						*BSRC1*			
cmpeqf.n	s		00		0	1	1	0011				000000						*BDEST 2*			*SRC2*						*SRC1*					
norl	s		00		0	1	1	0011				000001						*BDEST 2*					*BSRC2*						*BSRC1*			
cmpgtf.n	s		00		0	1	1	0100				000000						*BDEST 2*			*SRC2*						*SRC1*					

Table 164. Instruction encoding (Continued)

	31	30	29 28	27	26	25	24 23 22 21	20 19 18 17 16 15	14 13 12	11 10 9 8 7 6	5 4 3 2 1 0
cmple	s		00	0	1	1	0110	000000	*BDEST 2*	*SRC2*	*SRC1*
cmpleu	s		00	0	1	1	0111	000000	*BDEST 2*	*SRC2*	*SRC1*
cmplt	s		00	0	1	1	1000	000000	*BDEST 2*	*SRC2*	*SRC1*
cmpeq.ph	s		00	0	1	1	1 000	001000	*BDEST 2*	*SRC2*	*SRC1*
cmpltu	s		00	0	1	1	1001	000000	*BDEST 2*	*SRC2*	*SRC1*
cmpgt.ph	s		00	0	1	1	1 001	001000	*BDEST 2*	*SRC2*	*SRC1*
andl	s		00	0	1	1	1010	000000	*BDEST 2*	*SRC2*	*SRC1*
nandl	s		00	0	1	1	1011	000000	*BDEST 2*	*SRC2*	*SRC1*
orl	s		00	0	1	1	1100	000000	*BDEST 2*	*SRC2*	*SRC1*
norl	s		00	0	1	1	1101	000000	*BDEST 2*	*SRC2*	*SRC1*
mov	s		00	0	1	1	1110	000000	*BDEST 2*	000000	*SRC1*
add	s		00	1	0		00000	*ISRC2*		*IDEST*	*SRC1*
sub	s		00	1	0		00001	*ISRC2*		*IDEST*	*SRC1*
shl	s		00	1	0		00010	*ISRC2*		*IDEST*	*SRC1*
shr	s		00	1	0		00011	*ISRC2*		*IDEST*	*SRC1*
shru	s		00	1	0		00100	*ISRC2*		*IDEST*	*SRC1*
sh1add	s		00	1	0		00101	*ISRC2*		*IDEST*	*SRC1*
sh2add	s		00	1	0		00110	*ISRC2*		*IDEST*	*SRC1*
sh3add	s		00	1	0		00111	*ISRC2*		*IDEST*	*SRC1*
addpc	s		00	1	0		01000	*ISRC2*		*IDEST*	000000
and	s		00	1	0		01001	*ISRC2*		*IDEST*	*SRC1*
andc	s		00	1	0		01010	*ISRC2*		*IDEST*	*SRC1*
or	s		00	1	0		01011	*ISRC2*		*IDEST*	*SRC1*
xor	s		00	1	0		01101	*ISRC2*		*IDEST*	*SRC1*
clz	s		00	1	0		01110	000000100		*IDEST*	*SRC1*
mul64h	s		00	1	0		01111	*ISRC2*		*NLIDEST*	*SRC1*
max	s		00	1	0		10000	*ISRC2*		*IDEST*	*SRC1*
maxu	s		00	1	0		10001	*ISRC2*		*IDEST*	*SRC1*
min	s		00	1	0		10010	*ISRC2*		*IDEST*	*SRC1*
minu	s		00	1	0		10011	*ISRC2*		*IDEST*	*SRC1*
mul64hu	s		00	1	0		10100	*ISRC2*		*NLIDEST*	*SRC1*
mul32	s		00	1	0		10110	*ISRC2*		*NLIDEST*	*SRC1*
mulll	s		00	1	0		11001	*ISRC2*		*NLIDEST*	*SRC1*
mulllu	s		00	1	0		11010	*ISRC2*		*NLIDEST*	*SRC1*
mulfrac	s		00	1	0		11111	*ISRC2*		*NLIDEST*	*SRC1*
cmpeq	s		00	1	1	0	0000	*ISRC2*		*IDEST*	*SRC1*
cmpne	s		00	1	1	0	0001	*ISRC2*		*IDEST*	*SRC1*

Table 164. Instruction encoding (Continued)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cmpge	s		00		1	1	0	0010				*ISRC2*									*IDEST*						*SRC1*					
cmpgeu	s		00		1	1	0	0011				*ISRC2*									*IDEST*						*SRC1*					
cmpgt	s		00		1	1	0	0100				*ISRC2*									*IDEST*						*SRC1*					
cmpgtu	s		00		1	1	0	0101				*ISRC2*									*IDEST*						*SRC1*					
cmple	s		00		1	1	0	0110				*ISRC2*									*IDEST*						*SRC1*					
cmpleu	s		00		1	1	0	0111				*ISRC2*									*IDEST*						*SRC1*					
cmplt	s		00		1	1	0	1000				*ISRC2*									*IDEST*						*SRC1*					
cmpltu	s		00		1	1	0	1001				*ISRC2*									*IDEST*						*SRC1*					
cmpeq	s		00		1	1	1	0000				*ISRC2*												*IBDEST*			*SRC1*					
cmpne	s		00		1	1	1	0001				*ISRC2*												*IBDEST*			*SRC1*					
cmpge	s		00		1	1	1	0010				*ISRC2*												*IBDEST*			*SRC1*					
cmpgeu	s		00		1	1	1	0011				*ISRC2*												*IBDEST*			*SRC1*					
cmpgt	s		00		1	1	1	0100				*ISRC2*												*IBDEST*			*SRC1*					
cmpgtu	s		00		1	1	1	0101				*ISRC2*												*IBDEST*			*SRC1*					
cmple	s		00		1	1	1	0110				*ISRC2*												*IBDEST*			*SRC1*					
cmpleu	s		00		1	1	1	0111				*ISRC2*												*IBDEST*			*SRC1*					
cmplt	s		00		1	1	1	1000				*ISRC2*												*IBDEST*			*SRC1*					
cmpltu	s		00		1	1	1	1001				*ISRC2*												*IBDEST*			*SRC1*					
shls	s		01		0	0	0	0	000			000			*NLDEST*						*SRC2*						*SRC1*					
shlso	s		01		0	0	0	0	001			000			*NLDEST*						*SRC2*						*SRC1*					
sxt	s		01		0	0	0	0	101			000			*DEST*						*SRC2*						*SRC1*					
zxt	s		01		0	0	0	0	110			000			*DEST*						*SRC2*						*SRC1*					
rotl	s		01		0	0	0	0	111			000			*DEST*						*SRC2*						*SRC1*					
perm.pb	s		01		0	0	1	0	000			000			*DEST*						*SRC2*						*SRC1*					
shuff.pbh	s		01		0	0	1	0	000			001			*DEST*						*SRC2*						*SRC1*					
sadu.pb	s		01		0	0	1	0	000			010			*NLDEST*						*SRC2*						*SRC1*					
shuff.pbl	s		01		0	0	1	0	001			001			*DEST*						*SRC2*						*SRC1*					
absubu.pb	s		01		0	0	1	0	001			010			*DEST*						*SRC2*						*SRC1*					
shuffodd.pb	s		01		0	0	1	0	010			001			*DEST*						*SRC2*						*SRC1*					
muladdus.pb	s		01		0	0	1	0	010			010			*NLDEST*						*SRC2*						*SRC1*					
shuffeve.pb	s		01		0	0	1	0	011			001			*DEST*						*SRC2*						*SRC1*					
pack.pb	s		01		0	0	1	0	011			010			*DEST*						*SRC2*						*SRC1*					
ext1.pb	s		01		0	0	1	0	100			001			*DEST*						*SRC2*						*SRC1*					
ext2.pb	s		01		0	0	1	0	101			001			*DEST*						*SRC2*						*SRC1*					
ext3.pb	s		01		0	0	1	0	110			001			*DEST*						*SRC2*						*SRC1*					
packsu.pb	s		01		0	0	1	0	111			001			*DEST*						*SRC2*						*SRC1*					
shl.ph	s		01		0	0	1	1	000			000			*DEST*						*SRC2*						*SRC1*					
add.ph	s		01		0	0	1	1	000			001			*DEST*						*SRC2*						*SRC1*					
shuff.phh	s		01		0	0	1	1	000			010			*DEST*						*SRC2*						*SRC1*					
shls.ph	s		01		0	0	1	1	001			000			*NLDEST*						*SRC2*						*SRC1*					
abss.ph	s		01		0	0	1	1	001			001			*DEST*						000000						*SRC1*					
shuff.phl	s		01		0	0	1	1	001			010			*DEST*						*SRC2*						*SRC1*					
shr.ph	s		01		0	0	1	1	010			000			*DEST*						*SRC2*						*SRC1*					
max.ph	s		01		0	0	1	1	010			001			*DEST*						*SRC2*						*SRC1*					
sub.ph	s		01		0	0	1	1	010			010			*DEST*						*SRC2*						*SRC1*					

Table 164. Instruction encoding (Continued)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
shrrnp.ph	s		01		0	0	1	1	011			000			*NLDEST*						*SRC2*						*SRC1*					
min.ph	s		01		0	0	1	1	011			001			*DEST*						*SRC2*						*SRC1*					
mulfracadds.ph	s		01		0	0	1	1	011			010			*NLDEST*						*SRC2*						*SRC1*					
shrrne.ph	s		01		0	0	1	1	100			000			*NLDEST*						*SRC2*						*SRC1*					
mul.ph	s		01		0	0	1	1	100			001			*NLDEST*						*SRC2*						*SRC1*					
muladd.ph	s		01		0	0	1	1	101			001			*NLDEST*						*SRC2*						*SRC1*					
mulfracrm.ph	s		01		0	0	1	1	110			001			*NLDEST*						*SRC2*						*SRC1*					
packrnp.phh	s		01		0	0	1	1	110			010			*DEST*						*SRC2*						*SRC1*					
mulfracrne.ph	s		01		0	0	1	1	111			001			*NLDEST*						*SRC2*						*SRC1*					
packs.ph	s		01		0	0	1	1	111			010			*DEST*						*SRC2*						*SRC1*					
addcg	s		01		0	1	00		*SCOND*			*BDEST*			*DEST*						*SRC2*						*SRC1*					
imml	s		01		0	1	01		0	*IMM*																						
immr	s		01		0	1	01		1	*IMM*																						
slct.pb	s		01		0	1	10		*SCOND*			000			*DEST*						*SRC2*						*SRC1*					
slct	s		01		0	1	10		*SCOND*			001			*DEST*						*SRC2*						*SRC1*					
mov	s		01		0	1	11		*SCOND*			001			*DEST*						000000						000000					
extl.pb	s		01		0	1	11		*SCOND*			010			*DEST*						*SRC2*						*SRC1*					
extr.pb	s		01		0	1	11		*SCOND*			011			*DEST*						*SRC2*						*SRC1*					
avgu.pb	s		01		0	1	11		*SCOND*			100			*DEST*						*SRC2*						*SRC1*					
avg4u.pb	s		01		0	1	11		*SCOND*			101			*NLDEST*						*SRC2*						*SRC1*					
shls	s		01		1	0	0	0	000			*ISRC2*									*NLIDEST*						*SRC1*					
shlso	s		01		1	0	0	0	001			*ISRC2*									*NLIDEST*						*SRC1*					
shrrnp	s		01		1	0	0	0	010			*ISRC2*									*NLIDEST*						*SRC1*					
extract	s		01		1	0	0	0	011			*ISRC2*									*IDEST*						*SRC1*					
extractu	s		01		1	0	0	0	100			*ISRC2*									*IDEST*						*SRC1*					
sxt	s		01		1	0	0	0	101			*ISRC2*									*IDEST*						*SRC1*					
zxt	s		01		1	0	0	0	110			*ISRC2*									*IDEST*						*SRC1*					
rotl	s		01		1	0	0	0	111			*ISRC2*									*IDEST*						*SRC1*					
extractl	s		01		1	0	0	1	011			*ISRC2*									*IDEST*						*SRC1*					
extractlu	s		01		1	0	0	1	100			*ISRC2*									*IDEST*						*SRC1*					
perm.pb	s		01		1	0	1	0	000			*ISRC2*									*IDEST*						*SRC1*					
shl.ph	s		01		1	0	1	1	000			*ISRC2*									*IDEST*						*SRC1*					
shls.ph	s		01		1	0	1	1	001			*ISRC2*									*NLIDEST*						*SRC1*					
shr.ph	s		01		1	0	1	1	010			*ISRC2*									*IDEST*						*SRC1*					
shrrnp.ph	s		01		1	0	1	1	011			*ISRC2*									*NLIDEST*						*SRC1*					
shrrne.ph	s		01		1	0	1	1	100			*ISRC2*									*NLIDEST*						*SRC1*					
slct	s		01		1	1	00		*SCOND*			*ISRC2*									*IDEST*						*SRC1*					
slctf	s		01		1	1	01		*SCOND*			*ISRC2*									*IDEST*						*SRC1*					
slct.pb	s		01		1	1	10		*SCOND*			*ISRC2*									*IDEST*						*SRC1*					
slctf.pb	s		01		1	1	11		*SCOND*			*ISRC2*									*IDEST*						*SRC1*					
ldl	s		10		0000				000			*ISRC2*									*IDESTP*						*SRC1*					
ldlc	s		10		0000				*PCOND*			*ISRC2*									*IDESTP*						*SRC1*					
pft	s		10		0000				000			*ISRC2*									000000						*SRC1*					
pftc	s		10		0000				*PCOND*			*ISRC2*									000000						*SRC1*					

Table 164. Instruction encoding (Continued)

	31	30	29 28	27 26 25 24	23 22 21	20 19 18 17 16 15 14 13 12	11 10 9 8 7 6	5 4 3 2 1 0
ldw	s		10	0001	000	*ISRC2*	*IDEST*	*SRC1*
ldwc	s		10	0001	*PCOND*	*ISRC2*	*IDEST*	*SRC1*
ldh	s		10	0010	000	*ISRC2*	*NLIDEST*	*SRC1*
ldhc	s		10	0010	*PCOND*	*ISRC2*	*NLIDEST*	*SRC1*
ldhu	s		10	0011	000	*ISRC2*	*NLIDEST*	*SRC1*
ldhuc	s		10	0011	*PCOND*	*ISRC2*	*NLIDEST*	*SRC1*
ldb	s		10	0100	000	*ISRC2*	*NLIDEST*	*SRC1*
ldbc	s		10	0100	*PCOND*	*ISRC2*	*NLIDEST*	*SRC1*
ldbu	s		10	0101	000	*ISRC2*	*NLIDEST*	*SRC1*
ldbuc	s		10	0101	*PCOND*	*ISRC2*	*NLIDEST*	*SRC1*
stl	s		10	1000	000	*ISRC2*	*SRC2P*	*SRC1*
stlc	s		10	1000	*PCOND*	*ISRC2*	*SRC2P*	*SRC1*
stw	s		10	1001	000	*ISRC2*	*SRC2*	*SRC1*
stwc	s		10	1001	*PCOND*	*ISRC2*	*SRC2*	*SRC1*
sth	s		10	1010	000	*ISRC2*	*SRC2*	*SRC1*
sthc	s		10	1010	*PCOND*	*ISRC2*	*SRC2*	*SRC1*
stb	s		10	1100	000	*ISRC2*	*SRC2*	*SRC1*
stbc	s		10	1100	*PCOND*	*ISRC2*	*SRC2*	*SRC1*
prgadd	s		10	1111	000	*ISRC2*	000000	*SRC1*
prgadd.l1	s		10	1111	000	*ISRC2*	000001	*SRC1*
prgset	s		10	1111	001	*ISRC2*	000000	*SRC1*
prgset.l1	s		10	1111	001	*ISRC2*	000001	*SRC1*
prginsadd	s		10	1111	010	*ISRC2*	000000	*SRC1*
prginsadd.l1	s		10	1111	010	*ISRC2*	000001	*SRC1*
prginsset	s		10	1111	011	*ISRC2*	000000	*SRC1*
prginsset.l1	s		10	1111	011	*ISRC2*	000001	*SRC1*
flushadd	s		10	1111	100	*ISRC2*	000000	*SRC1*
flushadd.l1	s		10	1111	100	*ISRC2*	000001	*SRC1*
pswmask	s		10	1111	101	*ISRC2*	*NLIDEST*	*SRC1*
invadd	s		10	1111	110	*ISRC2*	000000	*SRC1*
invadd.l1	s		10	1111	110	*ISRC2*	000001	*SRC1*
syncins	s		10	1111	111	000000000	000000	000000
sync	s		10	1111	111	000000010	000000	000000
wmb	s		10	1111	111	000000011	000000	000000
waitl	s		10	1111	111	000000100	000000	000000
ldwl	s		10	1111	111	000000101	*IDEST*	*SRC1*
dbgsbrk	1		10	1111111		000001000	*BRKNUM*	
sbrk	1		10	1111111		000001001	*BRKNUM*	
syscall	s		10	1111111		000001010	*BRKNUM*	
break	s		10	1111111		000001011	000000000000	
retention	1		10	1111	111	000001100	000000	000000
stwl	s		10	1111	111	100000 *BDEST 2*	*SRC2*	*SRC1*
call	s		11	0 000	0	*BTARG*		
call	s		11	0 000	1	00000000000000000000000		
goto	s		11	0 001	0	*BTARG*		

Table 164. Instruction encoding (Continued)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
goto	s		11		0	001			1	00000000000000000000000																						
rfi	s		11		0	010			0	00000000000000000000000																						
return	s		11		0	011			1	00000000000000000000000																						
br	s		11		1	0	*BCOND*			*BTARG*																						
brf	s		11		1	1	*BCOND*			*BTARG*																						

Appendix B STBus endian behavior

The processor behaves in a different manner depending to whether the ST240 is operating in big endian or little endian mode. *Section 23.5.2: Memory model on page 215* introduces the notation used in this appendix and defines the operation of the ST240 in terms of a logical view of memory. This appendix describes the mapping between that logical memory and an actual physical memory attached to an STBus.

B.1 Endianness of bytes and half-words within a word based memory

The STBus views memory as being constructed from an array of 32-bit words. The notation WMEM[i] is used to represent 32-bit words in memory where i varies in the range [0, 2^{30}), and MEM[s] represents a byte indexed within WMEM[i].

For a little endian memory system:

$$MEM[s] = WMEM[s/4]_{<8(s\backslash 4) \text{ FOR } 8>}$$

For a big endian memory system:

$$MEM[s] = WMEM[s/4]_{<8(3-s\backslash 4) \text{ FOR } 8>}$$

Half-word accesses are made by pairing byte accesses using the equations given above.

Considering two processors of different endianness connected to the same memory system, and representing the logical memory as seen by them as MEMLE[i] for the little endian processor and MEMBE[i] for the big endian processor:

$$MEM_{LE}[i] = MEM_{BE}[i \oplus 3]$$

and:

$$WMEM_{LE}[i] = WMEM_{BE}[i]$$

As an example given the word WMEM[i], which stores the value 0xAABBCCDD. In either endianness the word will read the same, but when read as bytes by a little endian processor:

$MEM_{LE}[i]$ = 0xDD

$MEM_{LE}[i+1]$ = 0xCC

$MEM_{LE}[i+2]$ = 0xBB

$MEM_{LE}[i+3]$ = 0xAA

When read by a big endian processor:

$MEM_{BE}[i]$ = 0xAA

$MEM_{BE}[i+1]$ = 0xBB

$MEM_{BE}[i+2]$ = 0xCC

$MEM_{BE}[i+3]$ = 0xDD

B.2 Endianness of 64-bit accesses

The ST240 has a 64-bit STBus initiator port. The data presented to the STBus is determined differently depending upon the endianness mode. The STBus also interprets the information differently.

DMEM[i] refers to a double word in memory where i varies in the range [0, 2^{29}). When a little endian processor accesses a word address s:

$WMEM_{LE}[s] = DMEM_{LE}[s/2]_{<32(s\backslash 2) \text{ FOR } 32>}$

and for a big-endian processor:

$WMEM_{BE}[s] = WMEM_{LE}[s/2]_{<32(1-s\backslash 2) \text{ FOR } 32>}$

For example, if

WMEM[*s*]=0xaaaaaaaa

WMEM[*s*+1]=0xbbbbbbbb

then the contents of DMEM are:

$DMEM_{LE}[s]$ = 0xbbbbbbbb_aaaaaaaa

$DMEM_{BE}[s]$ = 0xaaaaaaaa_bbbbbbbb

the order of words in the double word has changed.

B.3 System requirements

Systems operating purely in a single mode are straightforward. All accesses as seen by the processor are consistent and behave as would be expected for a processor of that endianness.

Issues can arise where the memory system can be observed in both little-endian and big-endian modes. A correctly implemented system behaves according to the definitions given in this document. To ensure a correct implementation, the following points must be addressed in the system.

- The STBus and all devices with 64-bit target ports must be aware of the endianness of an access.
- Size convertors must be correctly configured for the endianness of the system. The correct operation of any size convertors ensure that 32-bit target ports do not need to be aware of endianness.

Danger: **If a system is NOT properly configured then the problems listed below may occur.**

- The peripheral registers of the ST240 may appear at the wrong address; bit 2 of the address could be inverted. This can be caused by a size convertor not being aware of endianness.
- Pairs of words may be swapped in memory.
- Words may be written to the wrong address; bit 2 could be inverted.

Glossary

Branch registers	The set of eight 4-bit registers which are used to store the condition for conditiona branches, carry bits, select operations, and other SIMD and logical operations.
Bundle	Wide instruction of multiple operations issued during the same cycle and executed in parallel.
Cache set	A set of a cache refers to all cache lines which may contain data at a given address. For a direct mapped cache the size of the set is 1, and for an *n*-way set associative cache the size of the set is *n*.
Commit point	The point at which the results of operations are written to the architectural state of the ST240.
Control register	One of a set of memory mapped registers maintained by the hardware or by software which form part of the architectural state.
Core	The core is the ST240 processor core excluding peripherals.
DTCM	Data-side Tightly Coupled Memory. Memory mapped RAM accessed in parallel with the data cache.
Dyadic operation	An operation on two operands.
General purpose registers	The set of directly addressed general purpose registers. The ST240 contains one general purpose register file organized as a bank of 64 32-bit registers.
Half-word, word, long word	Half-word relates to a 16-bit data item. Word relates to a 32-bit data item. Long word relates to a 64-bit data item.
If conversion	The transformation of the code generated by a compiler from an IF statement in a high level language into a sequence of bundles which do not include branches.
Immediate	An operand which is a constant or literal value.
Jump	A jump causes an unconditional changed in program counter to the destination address specified within the encoding. This is included within the ST200 architecture as a **goto** operation.
Level 1 cache	The Level- cache is also referred as the “closest” or “lowest” cache. Such a cache is tightly coupled within the core.
Level 2 cache	The Level 2 cache is a cache which is loosely coupled to the core. Typically used in cases where the level 1 cache or caches are insufficient in size or associativity. The level 2 cache is accessed through the level 1 cache or caches.
Long word	See half-word definition.
LRU	Least Recently Used. A replacement policy for caches and buffers. An LRU policy will replace the oldest entry whenever there is insufficient space for a new entry.
Main memory	This is the system-accessible memory.

Misaligned	A memory access is misaligned if the access does not fit the natural alignment width of the word being accessed, and the access is illegal.
Monadic operation	An operation on one operand.
Operation	An operation is an atomic ST200 action. An operation is equivalent to a typical instruction of a traditional 32-bit RISC machine.
Predication	The operation of selectively quashing an operation according to the value of a register (called predicate). The examples of this supported by the ST240 are select and conditional load/store operations.
Round robin	A replacement policy for caches and buffers. A round robin policy replaces entries in turn whenever there is insufficient space for a new entry.
Set	See cache set definition.
SIMD	Single Instruction Multiple Data. A class of operations which simultaneously performs the same operation on multiple bit fields of the source operand or operands.
ST240	The ST240 is the processor core as described in this manual including the associated peripherals. Also see “core” definition.
Superscalar	An architecture with multiple functional units in which instructions are scheduled dynamically by the hardware at run-time.
Syllable	Encoded component of a bundle that specifies one operation or immediate extension to an operation which encodes an immediate form. A bundle in the ST240 may contain multiple syllables, each of them 32-bit wide.
Unaligned	A memory access is unaligned if the access does not fit the natural alignment width of the word being accessed, but the access is still legal.
Uniprocessor	A single processor which is not part of a multi-processor cache coherent system.
VLIW	Very long instruction word: instructions (called “bundles” in ST200 terminology) encode one or more independent operations and are scheduled at compile time.
Word	See half-word definition.

List of instructions

Revision history

Table 165. Document revision history

Date	Revision	Changes
27-Jun-2008	C	Minor changes made throughout.
8-Feb-2008	B	Significant architectural changes have been made. A new chapter, *Chapter 9: SIMD operations on page 64*, has been added.
18-Sep-2007	A	Initial release.

Index

I

J

L

M

N

O

P

R

S

T

U

V

W

XYZ

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com