ST200 VLIW Series

ST220 Core and Instruction Set Architecture Manual

Last updated 12 September 2007

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.

The ST220 core is based on technology jointly developed by Hewlett-Packard Laboratories and STMicroelectronics

© 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007 STMicroelectronics - All rights reserved

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

Contents

Preface **xi**

1 Introduction **13**

1.1 VLIW overview 14
1.2 ST220 overview 14
1.3 Document overview 16

2 Cluster **19**

2.1 Integer units 20
2.2 Multiply units 21
2.3 Load/store unit (LSU) 21
2.3.1 Memory access 21
2.3.2 Addressing modes 22
2.3.3 Alignment 22
2.3.4 Control registers 22
2.3.5 Cache purging 22
2.3.6 Dismissible loads 22

3	**Architectural state**	**25**
3.1	Program counter (PC)	25
3.2	Register file	25
3.2.1	Link register	25
3.3	Program status word (PSW)	26
3.3.1	Bit fields	26
3.3.2	USER_MODE	27
3.3.3	PSW Access	27
3.3.4	Supported method for changing the PSW	27
3.4	Branch register file	28
3.5	Control registers	28
4	**Execution pipeline and latencies**	**29**
4.1	Execution pipeline	29
4.2	Operation latencies	30
4.3	Additional notes	30
4.3.1	Restrictions on link register	30
5	**Traps: exceptions and interrupts**	**31**
5.1	Trap mechanism	31
5.2	Exception handling	32
5.3	Saved execution state	32
5.4	Interrupts	34
5.5	Debug interrupt handling	34
5.6	Exception types and priorities	34
5.6.1	Illegal instruction definition	35
5.7	Speculative load considerations	36
5.7.1	Misaligned implementation	38
5.7.2	Speculative load exceptions	38

6 Memory access protection units 39

6.1 Description 39
6.2 Operation 40
6.2.1 Example use of overlapping regions 41
6.2.2 Undefined address space 42
6.3 Protection unit registers 42
6.3.1 Region base registers 42
6.3.2 Region attribute registers 43
6.3.3 Cacheable field 46
6.3.4 Speculative load returns zero field 46
6.3.5 Operation when protection unit is disabled 46

7 Memory subsystem 47

7.1 Memory subsystem 48
7.2 I-side memory subsystem 48
7.2.1 Instruction buffer 49
7.2.2 Instruction cache 49
7.2.3 I-side bus error 50
7.3 D-side memory subsystem 50
7.3.1 Load store unit 50
7.3.2 Cached loads and stores 50
7.3.3 Uncached load and stores 51
7.3.4 Prefetching data 51
7.3.5 Purging data caches 52
7.3.6 D-side synchronization 52
7.3.7 D-side bus errors 53
7.3.8 Operations 53
7.3.9 Write buffer 53
7.4 Core memory controller (CMC) 54

7.5 Additional notes 54
7.5.1 Forcing writes to external memory 54
7.5.2 Memory ordering 54
7.5.3 Coherency between I-side and D-side 55
7.5.4 Changing memory to uncacheable 55
7.5.5 Reset state 55
7.5.6 Cached data in uncached region 56
7.5.7 Prefetch performance 56

8 Streaming data interface (SDI) 57

8.1 Overview 57
8.2 Functional description 58
8.2.1 Data width 58
8.3 Communication channel 59
8.3.1 Timeouts 59
8.4 Registers 59
8.4.1 Input channel memory mapping 59
8.4.2 Output channel memory mapping 61
8.4.3 Protection 61
8.5 Exceptions, interrupts, reset and restart 62
8.5.1 Interrupts 62
8.5.2 SDI exceptions 63
8.5.3 Restart (or soft reset) 63

9 Control registers 65

9.1 Access operations 65
9.2 Exceptions 65
9.3 Control register addresses 66

10 Timers 71

10.1 Operation 72

10.1.1 TIMEDIVIDE 72
10.1.2 TIMECNTRi 72
10.1.3 TIMECNSTi 73
10.1.4 TIMECNTLi 73
10.1.5 TIMESTART 74

10.2 Timer interrupts 74

10.3 Programming the timer 74

11 Peripheral addresses 75

11.1 Peripheral addresses 75

11.1.1 Interrupt controller & timer registers 76
11.1.2 DSU registers 77
11.1.3 DSU ROM 78

12 Interrupt controller 79

12.1 Architecture 79

12.2 Operation 80

12.2.1 Test register 80

12.3 Interrupt registers 80

12.3.1 Interrupt pending register 80
12.3.2 Interrupt mask register (INTMASK) 81
12.3.3 Interrupt test register (INTTEST) 82

12.4 Programming 83

12.4.1 Enabling/disabling interrupts 83
12.4.2 Test register 83
12.4.3 Interrupt priority 83
12.4.4 Timer interrupts 83

13 Debugging support 85

13.1 Overview 85

13.2 Core 86

13.2.1 Debug interrupts 86

13.2.2 Hardware breakpoint support 87

13.3 Debug support unit 88

13.3.1 Architecture 88

13.3.2 Shared register bank 89

13.3.3 DSU control registers 90

13.4 Debug ROM 91

13.4.1 Debug initialization loop 91

13.4.2 Default debug handler 92

13.5 Host debug interface 95

13.5.1 Message format 95

13.5.2 Operation 97

14 Performance monitoring 99

14.1 Events 99

14.2 Access to registers 100

14.3 Control register (PM_CR) 101

14.4 Event counters (PM_CNTi) 102

14.5 Clock counter (PM_PCLK) 102

14.6 Recording events 102

15 Execution model 105

15.1 Introduction 105
15.2 Bundle fetch, decode, and execute 106
15.3 Functions 108
15.3.1 Bundle decode 108
15.3.2 Operation execution 108
15.3.3 Exceptional cases 108

16 Specification notation 109

16.1 Overview 109
16.2 Variables and types 110
16.2.1 Integer 110
16.2.2 Boolean 111
16.2.3 Bit-fields 111
16.2.4 Arrays 111
16.3 Expressions 112
16.3.1 Integer arithmetic operators 112
16.3.2 Integer shift operators 113
16.3.3 Integer bitwise operators 114
16.3.4 Relational operators 115
16.3.5 Boolean operators 116
16.3.6 Single-value functions 116
16.4 Statements 118
16.4.1 Undefined behavior 118
16.4.2 Assignment 118
16.4.3 Conditional 120
16.4.4 Repetition 121
16.4.5 Exceptions 121
16.4.6 Procedures 122
16.5 Architectural state 122

16.6 Memory and control registers 124

16.6.1 Support functions 124

16.6.2 Memory model 125

16.6.3 Control register model 131

16.6.4 Cache model 133

17 Instruction set 135

17.1 Introduction 135

17.2 Bundle encoding 135

17.2.1 Extended immediates 136

17.2.2 Encoding restrictions 137

17.3 Operation specifications 137

17.4 Example operations 139

17.4.1 add Immediate 139

17.5 Macros 141

17.6 Operations 141

A Instruction encoding 313

A.1 Reserved bits 313

A.2 Fields 313

A.3 Formats 314

A.4 Opcodes 316

Index 325

Preface

This document is part of the ST200 documentation suite detailed below. Comments on this or other manuals in the ST200 documentation suite should be made by contacting your local STMicroelectronics Limited sales office or distributor.

ST200 document identification and control

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

ADCS *nnnnnnnx*

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

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

ST200 documentation suite

The ST200 documentation suite comprises the following volumes:

ST220 Core and Instruction Set Architecture Manual (ADCS 7395369)

This manual describes the architecture and instruction set of the ST220 implementation.

ST200 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. 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 Compiler Manual (ADCS 7508723)

This manual describes the software provided as part of the ST200 tools. It supports the development of ST200 applications for embedded systems. Applications can be developed in either a stand-alone environment, or under the OS21 real-time operating system.

ST200 Runtime Architecture Manual (ADCS 7521848)

This manual describes the common software conventions for the ST200 processor runtime 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 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.

Introduction

1

The 32-bit ST220 is a member of the ST200 family of cores.

This family of embedded processors use a scalable technology that allows variation in instruction 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 register file and functional unit clustering.
- Support for user/supervisor modes and memory protection.

1.1 VLIW overview

VLIW (very long instruction word) processors use a technique where instruction level parallelism is explicitly exposed to the compiler which must schedule operations to account for the operation latency.

RISC-like operations (syllables) are grouped into bundles (wide words). The operations in a bundle are issued simultaneously. In the ST200 family operations also complete simultaneously. While the delay between issue and completion is the same for all operations, some results are available for bypassing to subsequent operations prior to completion. This is discussed further in *Chapter 4: Execution pipeline and latencies on page 29*.

A hardware implementation of a VLIW is significantly simpler than a corresponding multiple issue superscaler CPU. This is due principally to the simplification of the operation grouping and scheduling hardware, all this complexity is moved to the instruction scheduling system (compiler and assembler) in the software toolchain.

1.2 ST220 overview

The ST220 is a single cluster member of the ST200 family (see *Figure 1 on page 15*).

Clustering is a technique in which functional units and registers are tightly coupled in groups called, *clusters*. Multiple clusters can be instantiated in a processor with a single program counter controlling execution. To date no ST200 varient contains more than one cluster. No further discussion is made in this document of multiple clusters.

Figure 1: ST220 processor diagram

1.3 Document overview

This manual describes the architecture and instruction set of the ST220 implementation. This section gives an outline of the following document.

The processor is made up of a number of functional units described in *Chapter 2: Cluster* which operate on data stored in the register files (*Chapter 3: Architectural state*). These functional units are pipelined and subject to explicit observable latencies (*Chapter 4: Execution pipeline and latencies*).

The handling of exceptions and interrupts are detailed in *Chapter 5: Traps: exceptions and interrupts*.

The ST220 accesses memory through the memory subsystem (*Chapter 7: Memory subsystem*) which has limited protection provided by a pair of simple protection units for instruction and data accesses (*Chapter 6: Memory access protection units*).

The ST220 has 4 SDI ports (*Chapter 8: Streaming data interface (SDI)*) which allow it to communicate rapidly with other devices and avoid cache pollution when processing large amounts of data.

Control of the devices is performed using the memory mapped control registers defined within the relevant chapters. The address of the control registers and PSW are detailed in *Chapter 9: Control registers*.

The peripheral register addresses on the ST220 are detailed in *Chapter 11: Peripheral addresses on page 75*.

A number of peripheral devices are also provided, including timers (*Chapter 10: Timers*), interrupt control (*Chapter 12: Interrupt controller*) and debug support (*Chapter 13: Debugging support*).

The execution model is described in *Chapter 15: Execution model on page 105*. The execution of bundles is described in *Section 15.2: Bundle fetch, decode, and execute on page 106*, including the behavior of the machine when exceptions or interrupts are encountered.

Chapter 17 on page 135 describes the details of each operation, including the semantics. The instruction set includes details of the instruction set encoding, syntax and semantics. The encoding of bundles is defined in *Section 17.2: Bundle encoding on page 135*.

The behavior of operations is specified using the notational language defined in *Section 16.1* through *Section 16.4*. The descriptions clearly identify where architectural state is updated and the latency of the operands.

A simple model of memory and control registers defined in *Section 16.6.2* and *Section 16.6.3* is used when specifying the load and store operations.

Cluster

2

The ST220 cluster consists of the functional units and the two register files; the branch registers and general purpose registers (for register files, refer to *Chapter 3: Architectural state on page 25*). The cluster is the core of the processor but does not include the instruction fetch mechanism, caches or control registers.

This chapter describes the functional units of the ST220 cluster including 4 integer units, 2 multiply units, a load store unit and a branch unit (refer to *Figure 2*).

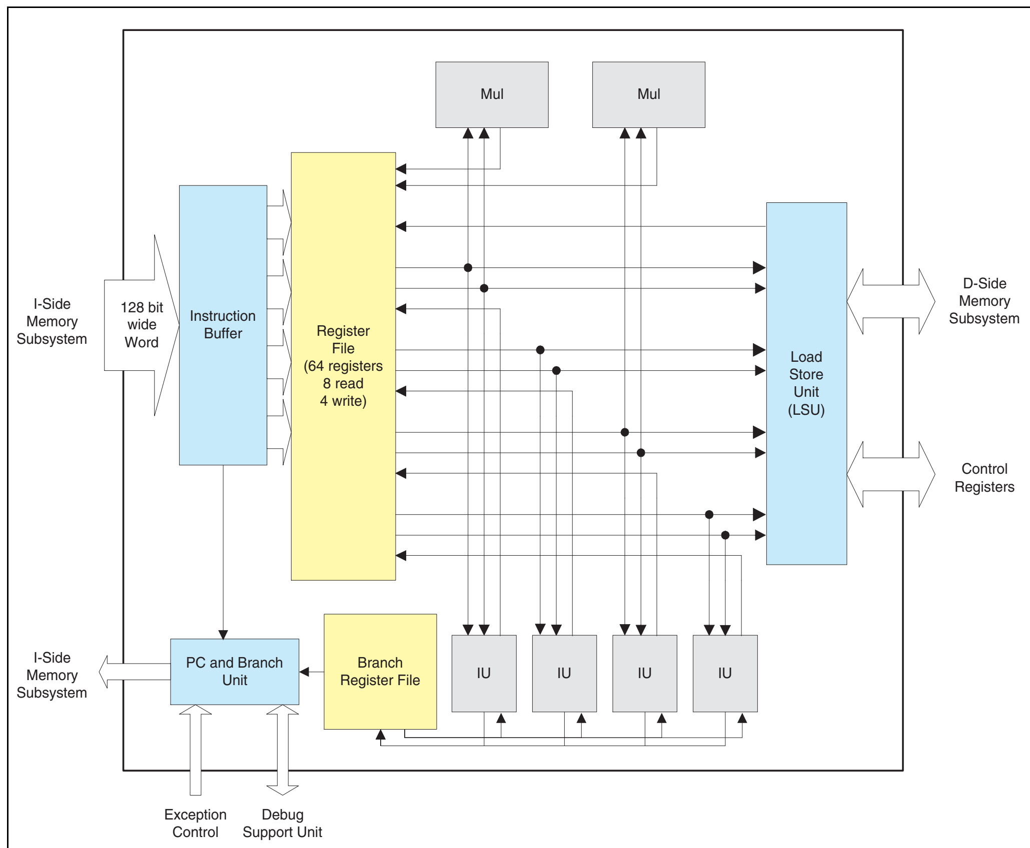


Figure 2: ST220 cluster

2.1 Integer units

The ST220 has four identical integer units. Each integer unit is capable of executing one operation per cycle. The results of the integer units can be used as operands of the next bundle. This is equivalent to a pipeline depth of one cycle.

Each operation can take up to three operands in the form of two 32-bit values and a single conditional bit. The IU then executes the appropriate operation and produces up to two results in the form of a 32-bit value and a 1-bit conditional value. The integer operations supported are detailed in the *Chapter 17: Instruction set on page 135*.

2.2 Multiply units

The ST220 has two identical multiply units. Each multiply unit is pipelined with a depth of three cycles, executing an operation every cycle.

Each multiply units takes two 32-bit operands and produces a single 32-bit result. The multiply operations supported are detailed in the *Chapter 17: Instruction set on page 135*.

2.3 Load/store unit (LSU)

The ST220 has a single load store unit. The load store unit is pipelined with a depth of three cycles, executing an operation every cycle.

The load store can take up to three 32-bit operands and may produce a single 32-bit result depending on the operation. The load store operations supported are detailed in the *Chapter 17: Instruction set on page 135*.

Memory access protection is implemented by the DPU (data protection unit), this is part of the memory sub-system. The DPU also controls the cache behavior of data accesses, *Chapter 6: Memory access protection units on page 39*.

Uncached accesses or accesses which miss the data cache cause the load store unit to stall the pipeline to ensure correct operation.

2.3.1 Memory access

The ST220 addresses the external memory system via a single address space. Peripheral devices and control registers are also mapped into the address space.

All cacheable memory transactions are made via the data cache. The data cache then decides if it needs to go to external memory to satisfy the request.

*Note: Cacheable memory transactions that miss are written to the write buffer **not** the data cache.*

Uncached accesses are performed directly on the memory system (refer to *Chapter 7*: *Section 7.3.3: Uncached load and stores on page 51*).

2.3.2 Addressing modes

The ST220 supports one addressing mode – the effective address is an immediate (constant) plus a register.

2.3.3 Alignment

1 All **load** and **store** instructions work on data stored on the "natural alignment" of the data type; that is, words on word boundaries, half-word on half word boundaries.

2 **Load** and **store** operations with misaligned addresses raise an exception which makes possible the implementation of misaligned **loads** by trap handlers.

3 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 instruction defintion.

4 For a byte or half-word **store**, the data stored from the least significant part of a register.

2.3.4 Control registers

The LSU maps a part of the address space that is devoted to control registers (see the Control Registers chapter for details). The LSU control register block intercepts **loads** and **stores** to this area of memory so that it can process the operation. No access to the data cache is made for control register operations. Transactions are made across the 32-bit control register bus to those control registers that live outside the LSU.

2.3.5 Cache purging

Cache purging (flush and invalidate) operations are provided on the ST220.

They allow for purging lines and sets from the data cache, and invalidating the entire instruction cache.

2.3.6 Dismissible loads

Dismissible **loads** are used to support software load speculation. This allows the compiler to schedule a **load** in advance of a condition that predicates its use.

Dismissible **loads** are required to return the same value as a normal **load** if such an operation can be executed without causing an exception. Otherwise dismissible **loads** return zero.

In the event that misaligned accesses are supported through a software trap handler, the ST220 may be configured to trap non-aligned dismissible **loads**, see the *Chapter 5: Traps: exceptions and interrupts on page 31*. The data protection unit can be configured to return zero for dismissible **loads** in cases where they can be executed without exception; this is to support peripherals which have destructive read behavior.

3 Architectural state

This chapter describes the architectural state of the ST220 core.

3.1 Program counter (PC)

The 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.

3.2 Register file

The general purpose register file contains 64 words of 32 bits, R0 ... R63.

Reading register zero, R0, always returns the value zero. Writing values to R0, has no effect on the processor state.

3.2.1 Link register

Register 63, R63, is the architectural link register used by the **call** and **return** mechanism. R63 is updated by explicit register writes and the **call** operation. Some restrictions apply to accessing the link register, see *Section 4.3.1: Restrictions on link register on page 30*.

3.3 Program status word (PSW)

The program status word (PSW) contains control information that affects the operation of the ST220 processor.

3.3.1 Bit fields

The PSW contains the following bit fields:

Name	Bit(s)	Access (U/S)	Reset	Comment
USER_MODE	0	RO/RW	0x0	When 1 the core is in user mode, otherwise supervisor mode.
INT_ENABLE	1	RO/RW	0x0	When 1 external interrupts are enabled.
Reserved	2	RO/RO	0x0	Reserved
Reserved	3	RO/RO	0x0	Reserved
SPECLOAD_MALIGNTRAP_EN	4	RO/RW	0x0	When 1 enables exceptions on speculative load misalignment errors.
SPECLOAD_DPUTRAP_EN	5	RO/RW	0x0	When 1 exceptions on speculative load DPU traps are enabled.
DPU_ENABLE	6	RO/RW	0x0	When 1 the DPU is enabled.
IPU_ENABLE	7	RO/RW	0x0	When 1 the IPU is enabled.
DBREAK_ENABLE	8	RO/RW	0x0	When 1 data breakpoints are enabled.
IBREAK_ENABLE	9	RO/RW	0x0	When 1 instruction breakpoints are enabled.
Reserved	10	RO/RO	0x0	Reserved
Reserved	11	RO/RO	0x0	Reserved
DEBUG_MODE	12	RO/RW	0x0	When 1 the core is in debug mode.
Reserved	[31:13]	RO/RO	0x0	Reserved

Table 1: PSW bit fields

Access to bits in the control register is restricted (read only or read write) dependant on the value of USER_MODE at the time of access.

3.3.2 USER_MODE

The USER_MODE bit indicates whether the machine is in **user** mode or **supervisor** mode. When in **user** mode, the processor has restricted access:

- The memory access protection units (see *Chapter 6: Memory access protection units on page 39*) define the level of access to memory in both **user** and **supervisor** modes.
- In **user** mode there is limited access to control registers (see *Chapter 9: Control registers on page 65*).
- Certain instructions can not be executed in **user** mode (see *Chapter 17: Instruction set on page 135*).

3.3.3 PSW Access

The PSW can be accessed as a control register, *Section 3.5: Control registers on page 28*.

3.3.4 Supported method for changing the PSW

To update the PSW correctly the following method, using the **rfi** operation, is recommended. The required status word should be stored into the SAVED_PSW and the address of the code to be executed directly after the change should be stored in the SAVED_PC. Then executing an **rfi** atomically copies the SAVED_PSW into the PSW and the SAVED_PC into the PC. At least four bundles are required to ensure that the changes to SAVED_PC and SAVED_PSW take effect before the **rfi** is executed.

Example: Procedure to write the PSW, (in ST220 assembler code),

```
_sys_set_psw:
   stw SAVED_PC[$r0.0] = $r0.63;; // Return address
   stw SAVED_PSW[$r0.0] = $r0.4;; // New value
   nop ;;
   nop ;;
   nop ;;
   nop ;;
   rfi ;;
```

Note: Interrupts must be disabled during this sequence.

3.4 Branch register file

The branch register file contains 8 single bit branch registers, B0 ... B7.

3.5 Control registers

Additional architectural state is held in a number of memory mapped control registers, *Chapter 9: Control registers on page 65*. These registers include support for interrupts and exceptions, and memory protection.

4 Execution pipeline and latencies

This chapter describes the architecturally visible pipeline and operation latencies.

4.1 Execution pipeline

The ST220 uses a pipelined execution scheme. This pipeline is architecturally visible in a number of areas:

- Operation latencies
- Bypassing
- Usage restrictions

The execution pipeline is three cycles long. It comprises of three stage E1, E2 and E3. All operations begin in E1. Operands are read or bypassed to an operation at the start of E1. All results are written at the end of E3.

This execution pipeline allows arithmetic and **load/store** operations to execute for up to three cycles. The results of operations which complete earlier than E3 are made available for bypassing as operands to subsequent operations, though strictly operations do not complete until the end of the E3 stage. This is when the architectural state is updated.

The pipeline is designed to efficiently implement the serial execution of the code (see *Chapter 15: Execution model on page 105*).

4.2 Operation latencies

ST220 operations begin in E1 cycle and complete in either E1, E2 or E3. The time taken for an operation to produce a result is called the operation latency. For simple operations like **add** and **subtract** the latency is a single cycle. For operations like **multiply** and **load** the latency is three cycles.

Note: Operational latencies may vary between different members of the ST200 processor family.

4.3 Additional notes

4.3.1 Restrictions on link register

As a performance optimization a speculative link register (SLR) has been added which is a copy of possible future updates to R63. In the implementation this register is updated earlier in the pipeline than R63. SLR is used as the source for register indirect branch operations.

It is possible to observe that SLR is not a true copy of R63. This can only occurs in the following cases;

- Register indirect **call** and **goto** operations. These require R63 to be stable. Solution; R63 must not be modified in the three bundles preceding one of these operations.
- The taking of an interrupt or exception just prior to an update of R63 but after SLR has been changed speculatively. Solution; All interrupt and exception handlers must explicitly write R63 prior to the execution of an **rfi**, **icall** or **igoto**. This requirement can easily be met with a **mov** operation from R63 to R63 in one of the first bundles of the trap handler.

A number of operations cannot target R63 for efficiency reasons. These include multiply operations, byte and half word load operations (see *Chapter 17: Instruction set on page 135*).

5 Traps: exceptions and interrupts

In the ST220 architecture, exceptions and interrupts are jointly termed traps. This chapter describes the trap mechanism.

5.1 Trap mechanism

The ST220 defines two types of traps:

- External asynchronous traps (interrupts).
- Internal synchronous traps (exceptions resulting from operation execution).

A trap point is the point in the program execution where a trap occurs. All bundles executed before the trap point will have completed updating architectural state; and no architectural state will have been undated by subsequent bundles. For an exception, the trap point is the (start of the) bundle which caused the exception. For an interrupt, the trap point is (the start of) the bundle whose execution has been interrupted. Typically this is a bundle that had been executed shortly after the interrupt was raised or enabled.

The flow diagram, *Figure 14* in *Section 15.2*, defines when a trap is taken. The aim of this chapter is to define the steps that are carried out when a trap is to be taken.

In effect, taking a trap can be viewed as executing an operation which branches to the required handler, with a number of side effects. The side effects are defined by the statements below. An external interrupt is treated as an EXTERN_INT exception, with only debug interrupts being handled differently.

At the trap point, the ST220 transfers execution to the trap handler, starting at the address held in the HANDLER_PC control register, and saves the execution state as detailed in *Table 5.3: Saved execution state* . All operations issued before the trapping bundle are allowed to complete. All operations issued after and including the trapping bundle are discarded. The architectural state, with the exception of saved execution state, is exactly that at the trap point. Hence ST220 interrupts and exceptions can be considered precise.

Traps are handled strictly (in order), and indivisibly with respect to the bundle stream.

5.2 Exception handling

Due to the fact that there may be more than one operation executing at once, it is possible to have more than one exception thrown in a bundle. However, only the highest priority exception is passed to the handler.

5.3 Saved 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 processor. Control registers are used to store these values for use by the handler routine.

Taking an exception can be summarized as:

```
NEXT_PC ←HANDLER_PC;    // Branch to the exception handler

EXCEPT_CAUSE ←HighestPriority();        // Store information
EXCEPT_ADDR ←DataAddress(EXCEPT_CAUSE);// for the handler

SAVED_PSW ←PSW;         // Save the PSW and PC
SAVED_PC ←BUNDLE_PC;

PSW[USER_MODE] ←0;      // Enter supervisor mode
PSW[INT_ENABLE] ←0;     // Disable interrupts
PSW[IBREAK_ENABLE] ←0;  // Disable instruction breakpoints
PSW[DBREAK_ENABLE] ←0;  // Disable data breakpoints
```

Where the function **HighestPriority** returns the highest priority exception from those that have been thrown (please refer to *Section 5.6*). The **DataAddress** function defines the value that is stored into the EXCEPT_ADDR control register. Its return value will either be **0** or the effective address of data which has triggered the exception.

Therefore,

```
variable ←DataAddress(exception);
is equivalent to:
IF ((exception = DBREAK) OR
     (exception = MISALIGNED_TRAP) OR
     (exception = CREG_NO_MAPPING) OR
     (exception = CREG_ACCESS_VIOLATION) OR
     (exception = DPU_NO_TRANSLATION) OR
     (exception = DPU_ACCESS_VIOLATION)) THEN
  variable ←value;
ELSE
  variable ←0;
```

Where **value** is the optional argument that will have been passed to the **THROW** (see *Section 16.4.5: Exceptions on page 121*) when the exception was generated.

The **rfi** (return from interrupt) operation is used to recommence execution at the trap point. An **rfi** operation will cause the following state updates:

```
PC ←SAVED_PC;                   // Address execution control is
                                // transferred to by rfi. Can be
                                // altered during the exception
                                // handler routine.

PSW ←SAVED_PSW;                 // Restore saved_psw. Can be
                                // altered during the exception
                                // handler routine.

SAVED_PC ←SAVED_SAVED_PC;       // Restore previous saved_pc

SAVED_PSW ←SAVED_SAVED_PSW;     // Restore previous saved_psw
```

5.4 Interrupts

All interrupts are effectively treated by the ST220 as an exception of type EXTERN_INT. Individual interrupt lines are indicated by registers in the interrupt controller, *Chapter 12: Interrupt controller on page 79*.

5.5 Debug interrupt handling

Please refer to *Chapter 13: Debugging support on page 85*.

5.6 Exception types and priorities

The table below shows the possible exceptions and the bit number in the EXCEPT_CAUSE control register that each corresponds to. Since only one exception is raised at a time, simultaneous exceptions are prioritized. The table is listed in exception priority order starting with the highest priority.

Name	Bit(s)	Access (U/S)	Reset	Comment
STBUS_IC_ERROR	0	RO/RW	0x0	The Instruction Cache caused a bus error.
STBUS_DC_ERROR	1	RO/RW	0x0	The Data Cache caused a bus error.
EXTERN_INT	2	RO/RW	0x0	There was an external interrupt.
IBREAK	3	RO/RW	0x0	The IPU has triggered a breakpoint on an instruction address.
IPU_NO_TRANSLATION	4	RO/RW	0x0	There was no mapping in the IPU for the given address.
IPU_ACCESS_VIOLATION	5	RO/RW	0x0	Permission to access an address controlled by the IPU was not met.
SBREAK	6	RO/RW	0x0	A software breakpoint was found.
ILL_INST	7	RO/RW	0x0	The bundle could not be decoded into legal sequence of operations or a privileged operation is being issued in user mode.

Table 2: EXCEPT_CAUSE bit fields

Name	Bit(s)	Access (U/S)	Reset	Comment
DBREAK	8	RO/RW	0x0	The DPU has triggered a breakpoint on a data address.
MISALIGNED_TRAP	9	RO/RW	0x0	The address is misaligned and misaligned accesses are not supported.
CREG_NO_MAPPING	10	RO/RW	0x0	The load or store address was in control register space, but no control register exists at that exact address.
CREG_ACCESS_VIOLATION	11	RO/RW	0x0	A store to a control register was attempted whilst in user mode.
DPU_NO_TRANSLATION	12	RO/RW	0x0	There was no mapping in the DPU for the given address.
DPU_ACCESS_VIOLATION	13	RO/RW	0x0	Permission to access an address controlled by the DPU was not met.
SDI_TIMEOUT	14	RO/RW	0x0	One of the SDI interfaces timed out while being accessed.

Table 2: EXCEPT_CAUSE bit fields

5.6.1 Illegal instruction definition

An illegal instruction exception is caused when an illegal bundle is executed. A legal bundle and all syllables contained in it must conform to the restrictions as detailed in *Chapter 17: Instruction set on page 135*.

A legal bundle and all syllables contained in it must conform to the following:

- All syllables must be valid operations
- A bundle must have a **stop** bit that is, four zero **stop** bits are illegal.
- Unused opcode fields must be set to zero, including bit 30.
- Any **branch** or **call** operation must appear as the first syllable of a bundle.
- Multiply operations must appear at odd word addresses.
- Long immediate extensions must appear at even word addresses.
- Immediate extensions must associate with an operation that is in the same bundle and has an immediate format that can be extended.

- A privileged operation can only be executed in **supervisor** mode.
- Only one memory operation can be performed in each bundle.
- A **sync** operation must be alone in a bundle.
- An **sbrk** operation must have the stop bit set.
- Destination registers in a bundle have to be unique, with the exception of R0.
- **ldb**, **ldh** and **mul** operations must not have R63 as a destination register.

5.7 Speculative load considerations

Speculative (or dismissible) loads are defined such that they execute as normal loads except in the following cases:

1. The address is in a peripheral region where a speculative load may be destructive. This is indicated by the DPU[SPEC_ZERO]. In this case a zero is always returned and no access is made to the peripheral region.
2. A normal load would cause an exception. Generally, in this case, the load is considered to have been incorrectly speculated and the data will not utilized in the correct execution of the program. Zero is returned by default, the following two sub-sections detail the exceptions to this behavior.
3. If a dismissible load causes a bus error then a bus error exception is always raised. The DPU should always be set up to prevent dismissible loads from causing bus errors.

The overall speculative load behavior is summarized in *Table 3: Summary of a dismissible read of address “a” from memory* .

DPU[SPEC_ZERO]	PSW[SPECLOAD_ MISALIGN_EN]	PSW[SPECLOAD_ DPUTRAP_EN]	**Misaligned(a)**	**DPUNoTranslation(a)**	**ReadAccessViolation(a)**	**Result**
0	0	0	0	0	0	Data = ReadMemory(a)
			0	0	1	Data = 0
			0	1	x	Data = 0
			1	x	x	Data = 0
0	0	1	0	0	0	Data = ReadMemory(a)
			0	0	1	DPU_ACCESS_VIOLATION
			0	1	x	DPU_NO_TRANSLATION
			1	x	x	Data = 0
0	1	0	0	0	0	Data = ReadMemory(a)
			0	0	1	Data = 0
			0	1	x	Data = 0
			1	x	x	MISALIGNED_TRAP
0	1	1	0	0	0	Data = ReadMemory(a)
			0	0	1	DPU_ACCESS_VIOLATION
			0	1	x	DPU_NO_TRANSLATION
			1	x	x	MISALIGNED_TRAP
1	x	x	x	x	x	Data = 0

Table 3: Summary of a dismissible read of address “a” from memory

5.7.1 Misaligned implementation

Application or system software may require misalignment support, with misaligned accesses being correctly interpreted by the exception handler. To improve speculative load support for misaligned addresses, a control value PSW[SPECLOAD_MALIGNTRAP_EN] can be set which causes speculative loads to trap on misaligned addresses rather than returning zero.

5.7.2 Speculative load exceptions

In some cases the software system may require a speculative load to cause a DPU exception instead of returning zero. This can be indicated by setting the SPECLOAD_DPUTRAP_EN bit in the PSW control register.

Note: Misaligned is not treated as a DPU trap.

6 Memory access protection units

The ST220 contains two memory protection units, one for instruction fetches (IPU) and one for load and store accesses (DPU). These units restrict access to memory according to whether the processor is in **user** or **supervisor** mode (see *Section 3.3.2: USER_MODE on page 27*) and also controls cacheability of memory accesses and the operation of speculative loads. The protection units do not provide address translation.

Note: The DPU is not used for control register access.

6.1 Description

On reset the instruction and data protection units are disabled, they are enabled and disabled through the PSW control register (see *Section 3.3: Program status word (PSW) on page 26*). For further details on the behavior of the processor when a protection unit is disabled (see *Section 6.3.5: Operation when protection unit is disabled on page 46*).

Note: It is stongly recommended that the IPU and DPU are enabled as soon as possible to protect against the generation of bus errors. Speculative loads should not be used until the DPU is enabled.

The control registers of the protection units define a number of regions[1] of address space. Each region has information associated with it, including:

- base address of region,
- size of region,
- access permissions of the region.

Note: Refer to *IPU attribute registers on page 44* *for full details.*

The size of a region is a ‘power of two’ number of bytes between 4 Kbyte and 4 Gbyte. The base of a region must be aligned on a size of region boundary.

The DPU and IPU differ in the number of regions and the access permissions supported. The IPU supports four regions (0-3) and the DPU supports eight regions (0-7).

The protection units allow regions to overlap which makes it possible to achieve complex mappings with a limited number of regions. If an address falls within more than one region, the information from the highest numbered region is used by the protection unit.

The protection units implement memory access restrictions including supervisor and user modes. They do not provide support for memory translation.

6.2 Operation

When enabled, each protection unit works by checking the address against the entries it holds. There are a number of possible outcomes:

No mapping	There is no mapping for the address in the protection unit.
One region hit	The properties of the selected region are applied.
Multiple region hits	The properties of the highest priority region are used.

1. The DPU/IPU should be disabled before making changes to the regions.

6.2.1 Example use of overlapping regions

If 4 Kbyte of **supervisor** code and 28 Kbyte of **user** code need to be mapped into a 32 Kbyte area of memory, the number of regions required are:

Overlapping regions not supported	Overlapping regions supported
One 4 K byte region for **supervisor** code	One 4 K byte region for **supervisor** code
One 16 K byte region for **user** code	One 32 K byte region for **user** code
One 8 K byte region for **user** code	
One 4 K byte region for **user** code	

Table 4: Overlapping regions

Diagrammatically, this is shown in *Figure 3*.

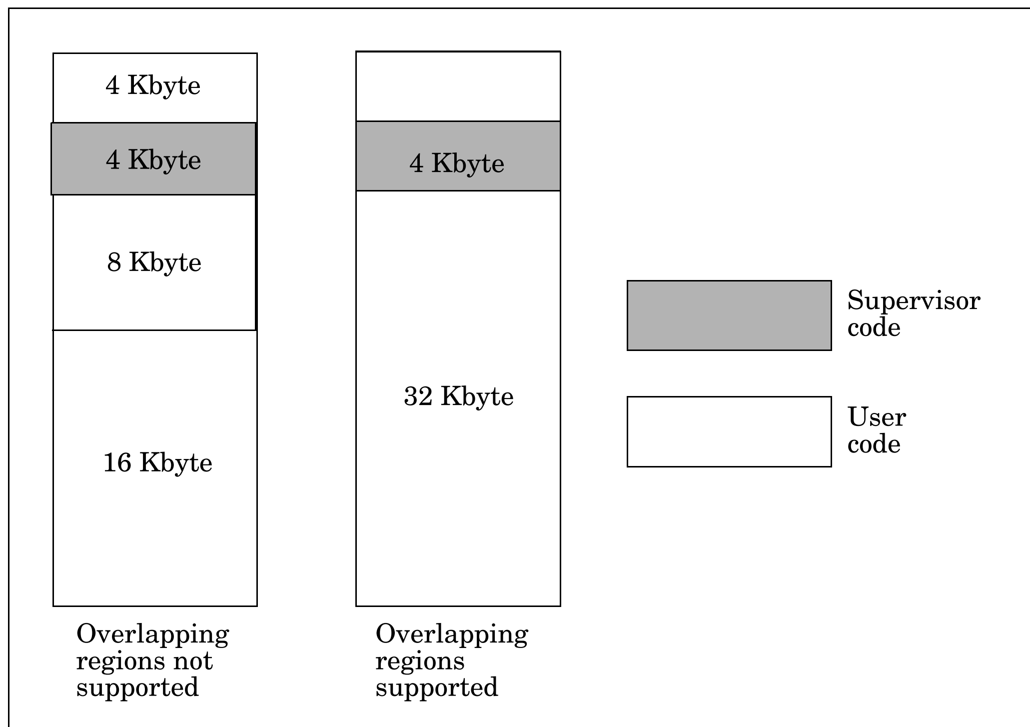


Figure 3: Overlapping regions

With overlapping regions, the 4 Kbyte of **supervisor** code would be put into a higher priority region. This ensures that the **supervisor** mapping takes precedence over the **user** mapping.

6.2.2 Undefined address space

The provision of overlapping regions of memory allows the default protection for otherwise unmapped memory to be programmed.

This can be done by setting region 0 to be 4 Gbyte in size, based at address 0. The behavior of otherwise unmapped memory can be defined, that is, setting permissions for no access in **supervisor** and **user** modes. By default an unmapped area of memory will create a ‘no translation’ exception for addresses with no translation.

6.3 Protection unit registers

6.3.1 Region base registers

Region base registers define the areas of memory that are controlled by the protection unit. Each region register contains a field to describe the base address of the memory region it is to cover.

Name	Bit(s)	Access (U/S)	Reset	Comment
REGION_ZERO	[11:0]	RO/RO	0x0	Always zero.
REGION_BASE	[31:12]	RO/RW	0x0	Region base.

Table 5: DPU_REGION0 bit fields

6.3.2 Region attribute registers

A summary of the fields within attribute registers is as follows:

- The OFFSET field gives the size of the region.
- Protection information (PROT) is encoded into two bits,
- The ENABLE bit enables the region when set to 1, otherwise the region is disabled.
- Cacheability is indicated by the CACHEABLE bit.
- The SPEC_ZERO bit indicates speculative load behavior in the region.

DPU attributes

Each DPU region has an attribute register that defines the properties and behavior of that region.

Name	Bit(s)	Access (U/S)	Reset	Comment
ENABLE	0	RO/RW	0x0	Enables this region.
PROT	[2:1]	RO/RW	0x0	Protection attributes of this region.
Reserved	[6:3]	RO/RO	0x0	Reserved
CACHEABLE	7	RO/RW	0x0	When set the region is cacheable.
SPEC_ZERO	8	RO/RW	0x0	When set speculative loads to this region return 0.
Reserved	[11:9]	RO/RO	0x0	Reserved
OFFSET	[16:12]	RO/RW	0x0	The size of this region.
Reserved	[31:17]	RO/RO	0x0	Reserved

Table 6: DPU_ATR0 bit fields

IPU attribute registers

The attributes of each IPU region are slightly different.

Name	Bit(s)	Access (U/S)	Reset	Comment
ENABLE	0	RO/RW	0x0	Enables this region.
PROT	[2:1]	RO/RW	0x0	Protection attributes of this region.
Reserved	[11:3]	RO/RO	0x0	Reserved
OFFSET	[16:12]	RO/RW	0x0	The size of this region.
Reserved	[31:17]	RO/RO	0x0	Reserved

Table 7: IPU_ATR0 bit fields

Offset field

The base of the region has to be aligned to the size of the region given in OFFSET (in the corresponding region attribute register). This restriction allows for faster computation of a region hit in the implementation. For example, the smallest region offset of 4 Kbyte must have a base address aligned to a 4 Kbyte boundary.

The offset value determines the size of the region, with the possible values given in the *Table 8*.

Offset[4:0]	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101
Region size	4 K byte	8 K byte	16 K byte	32 K byte	64 K byte	128 K byte	256 K byte	512 K byte	1 M byte	2 M byte	4 M byte

Offset[4:0]	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
Region size	8 M byte	16 M byte	32 M byte	64 M byte	128 M byte	256 M byte	512 M byte	1 G byte	2 G byte	4 G byte

Table 8: Offset field

An operating system (OS) can use large regions to set default permissions for the entire memory space.

The IPU has four regions. The four region and attribute registers are numbered from 0 to 3. If there are no matches between the given address and any described region an IPU_NO_TRANSLATION exception is generated.

The DPU has eight regions. The eight region and attribute registers, numbered from 0 to 7. If there are no matches between the given address and any described region a DPU_NO_TRANSLATION exception is generated.

DPU protection field

The DPU has the following encoding for its protection bits:

Value	Supervisor	User
00	No Access	No access
01	Read/write	No access
10	Read/write	Read only
11	Read/write	Read/write

The generation of the DPU_ACCESS_VIOLATION exception is summarized as follows:

Protection value	Memory access			
	Supervisor mode		User mode	
	Load	Store	Load	Store
00	Access violation	Access violation	Access violation	Access violation
01	OK	OK	Access violation	Access violation
10	OK	OK	OK	Access violation
11	OK	OK	OK	OK

IPU protection field

Value	Supervisor	User
00	Execute	Execute
01	No access	No access
10	Execute	No access
11	Reserved	

The generation of the IPU_ACCESS_VIOLATION is conditional upon whether the machine is in **supervisor** or **user** mode, on a region by region basis according to the above table.

6.3.3 Cacheable field

This field is used by the D-side memory subsystem to determine whether it may cache data from the region or not.

6.3.4 Speculative load returns zero field

In certain cases normal **loads** should return the data as expected but speculative **loads** to the same location should return zero without doing a memory access. If the SPEC_ZERO bit for the region is set in the attribute register, the LSU will be instructed to return zero as data and not make a memory request.

For a detailed description of how exceptions are handled with respect to this bit being set, see *Section 5.7: Speculative load considerations on page 36*.

6.3.5 Operation when protection unit is disabled

IPU Cached, no protection.

DPU Uncached, no protection. Speculative **loads** act as normal **loads**.

7 Memory subsystem

This chapter describes the operation of the ST220 processor memory subsystem. The memory subsystem includes the caches, protection units, write buffer, prefetch cache and the core memory controller (CMC).

The memory subsystem is split broadly into two parts, the instruction side (I-side) and the data side (D-side). The CMC interfaces these two halves to the STBus port. The I-side, containing the instruction cache and instruction protection unit (IPU), supports the fetching of instructions. The D-side, containing the data cache, data protection unit (DPU), prefetch cache and write buffer, support the storing and loading of data.

The ST220 ensures that data access are coherent with other data accesses. There is no guarantee of coherency between instruction and data accesses (see *Section 7.5.3: Coherency between I-side and D-side on page 55*) or between the core and external memory. To ensure coherency data must be purged from the core as decribed later in this chapter.

The functions of the IPU and DPU are described in detail in *Chapter 6: Memory access protection units on page 39*, the streaming data interface (SDI) in *Chapter 8: Streaming data interface (SDI) on page 57*.

7.1 Memory subsystem

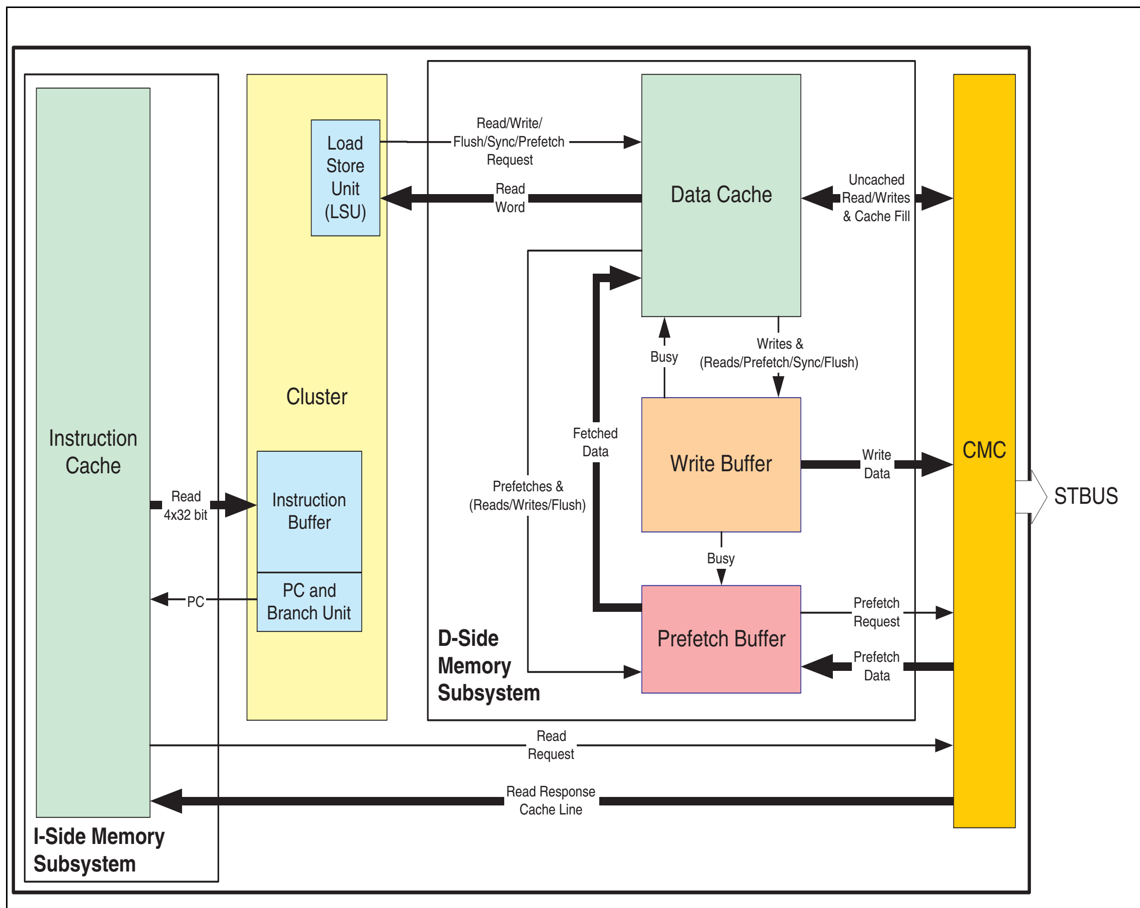


Figure 4: Memory subsystem block diagram

7.2 I-side memory subsystem

Within the ST220 the instruction buffer is responsible for issuing instructions to the processor core. The instruction cache fetches cache lines from memory, via the CMC and sends bundles of up to four operations to the instruction buffer.

7.2.1 Instruction buffer

The instruction buffer attempts to fetch ahead in the instruction stream in order to keep its buffer full. When a branch is taken the instruction buffer is invalidated and a fetch started from the target address.

After a branch the instruction buffer will take one cycle to fetch the next bundle from the cache, this means the ST220 will stall for one cycle. If the branch is to a bundle that spans two cache lines then it will take two cycles to fetch the bundle and thus the ST220 will stall for two cycles.

7.2.2 Instruction cache

Instructions are always cached; there is no support for uncached instruction fetching. Self modifying code (loaders for example) must invalidate the cache explicitly.

The instruction cache is a 32 Kbyte direct mapped cache with 64-byte lines. It receives fetch requests from the instruction buffer and returns a group of up to four 32-bit operations (16 bytes).

When instructions are requested from the cache it uses the address to determine whether they are already present in the cache. If the instructions are not in the cache, they are fetched from memory and stored into the cache, during which time the processor will stall. The requested instruction bundle is then returned to the instruction buffer.

To invalidate the instruction cache safely, two operations must be executed:

- The first is **prgins**, which invalidates the whole instruction cache and causes any subsequent instruction fetches to be made from memory rather than from the cache.
- The second is **syncins**, which ensure that all previous bundles have completed, and that their effects have completed, before any subsequent bundles are started. This guarantees that the next bundle will be fetched with an invalidated cache, and therefore from memory. If this operation is not performed the subsequent bundle might have been prefetched and might not correspond to the instruction in memory. Currently, the **syncins** operation is a pseudo operation implemented as a **goto** the next bundle.

7.2.3 I-side bus error

If the I-side memory subsystem causes a bus error, an STBUS_IC_ERROR exception is raised. Bus errors are asynchronous events and are not associated with a particular bundle.

In this case the cache will not be updated.

7.3 D-side memory subsystem

All data accesses take place through the D-side memory subsystem which contains the data cache, the prefetch cache and the write buffer. The data cache is 32 Kbyte 4-way associative with a 32-byte line. It is operated with a fixed write-back, no allocate on write-miss policy.

At most one of the write buffer, the data cache or the prefetch cache can contain a copy of the data for a particular address.

7.3.1 Load store unit

The load store unit (LSU) performs all data access operations. The cacheability is dependent on the address of the access and is determined by the DPU. In addition to **load** and **store** there are operations which prefetch data, flush and synchronize the D-side memory subsystem.

The data cache sends write misses and dirty data to the write buffer (see *Section 7.3.9: Write buffer on page 53*).

The write buffer combines write transactions and sends them out to memory.

7.3.2 Cached loads and stores

Cached **loads** and **stores** are performed through the data cache.

The memory subsystem can optimize these operations for performance.

For example, the memory subsystem can transfer more data than specified by the **load** (that is, a loading cache line), aggregate accesses (i.e. combining writes in write buffer) and/or re-order accesses (i.e. cache causes word accesses to be re-ordered).

The memory subsystem presents a consistent view of cached memory to the ST220 programmer, that is, a store followed by a load to the same address will always return the stored data. To guarantee ordering of accesses to external memory in cached regions, **purge** and **sync** operations must be used.

7.3.3 Uncached load and stores

Uncached **loads** and **stores** are performed directly on the STBus. Data from an uncached region of memory will never be brought into the data cache or prefetch cache. *Section 7.5.6: Cached data in uncached region on page 56*.

The precise amount of data specified in the access is transferred and the access is not aggregated with any other. The implementation does not optimize these accesses.

To guarantee that an uncached store has completed, either a sync or an uncached load to the same bus target must be issued.

7.3.4 Prefetching data

The prefetch cache prefetches and stores data from external memory and sends it to the data cache when (and if) it is required.

A **pft** operation is a hint to the memory subsystem that the given item of data may be accessed in the future. The operation specifies an address which can be prefetched by the prefetch cache. Prefetches have no effect on the functionality of the CPU but may change its performance. A **pft** operation may be ignored.

Prefetches to uncached areas, control registers and invalid addresses (that is, may cause a DPU exception) are treated as **nops**.

The prefetch cache contains eight entries. Each entry contains an entry valid bit, a prefetch address, a data valid bit and 32 bytes of data space.

When a **pft** request is made and accepted, it enters the prefetch cache as an outstanding prefetch request, with the data valid bit clear. Older entries may be discarded if the prefetch cache is full. The prefetch cache will attempt to access the memory system to fetch the line containing the prefetch address. When a fetch completes the data valid bit is set. The prefetch cache supports multiple outstanding memory requests.

Entries in the prefetch cache are tested when a data cache read miss occurs. If an entry match occurs and the data valid bit is set, the prefetched line is loaded into the data cache as if it were fetched from external memory. If the data valid bit is clear, the data cache stalls until the data is returned from external memory. The entry in the prefetch cache is then marked as empty and can be reused.

Entries in the prefetch cache are tested when a data cache write miss occurs. If an entry match occurs the prefetch cache entry is invalidated.

7.3.5 Purging data caches

The following purge (flush and invalidate) operations are used to ensure a copy of a particular data item is not cached in the D-side of the memory subsystem.

1. **prgadd** purges a line which hits the address operand from both caches (data and prefetch).
2. **prgset** purges the lines in the data cache set indicated by the address operand and purges the entire prefetch buffer.

These operations flush out the specified data. Dirty lines are written to the write buffer and the line is invalidated. Purge addresses are treated as byte aligned.

Purge operations cannot cause DPU exceptions.

7.3.6 D-side synchronization

This is achieved by executing the **sync** operation. Once the bundle containing the **sync** operation has completed, the following conditions hold:

1. All previous **loads**, **stores** and **pfts** have completed.
2. No future memory operations have started.
3. The write buffer is empty, all pending writes to external memory have completed.

7.3.7 D-side bus errors

If the D-side memory subsystem causes a bus error, a STBUS_DC_ERROR exception is raised. Bus errors are asynchronous events and are not associated with a particular operation.

In the case of writes the data will have already been discarded and therefore the write is lost. The write may or may not have completed.

In the case of reads the cache will not be updated.

7.3.8 Operations

The memory subsystem supports the following operations:

Type	Word aligned	Half word aligned		Byte aligned	
Load	Load word	Load half unsigned	Load half signed	Load byte unsigned	Load byte signed
Load Dismissible	Load word	Load half unsigned	Load half signed	Load byte unsigned	Load byte signed
Store	Store word	Store half		Store byte	
Prefetch				Prefetch	
Purge				Purge address	
				Purge set	
Sync				Sync	

It is a requirement that half word load/stores are half word aligned (2 bytes) and word **load/stores** are word aligned (4 bytes). Misaligned accesses will cause a MISALIGNED_TRAP exception.

7.3.9 Write buffer

Writes that miss the data cache and dirty lines that are evicted from the cache are held in the write buffer pending write back to external memory.

The write buffer is a write combining buffer that holds up to four entries. Each entry has 32 bytes of data, an address and 32 bits of byte masks. The write buffer is operated as an LRU (least recently used) buffer.

Write combining allows individual close proximity writes to be merged into a single entry. Write combining improves performance significantly (for a no write allocate cache) when performing sequences of writes to blocks of data which have not been brought into the cache.

7.4 Core memory controller (CMC)

The CMC allows multiple masters to access the STBus via a single port. The CMC arbitrates between multiple requestors and correctly routes responses.

7.5 Additional notes

The memory subsystem requires some additional explanation of some key operations and methods of use. This section is intended to provide this information without filling out the previous sections.

7.5.1 Forcing writes to external memory

After any purge operations have taken place a **sync** should be issued to ensure all writes, including cache write backs, have completed. This combination of operations will guarantee that the purged data will not be in the data cache, prefetch cache or write buffer.

The **sync** operation should also be used to ensure that uncached writes have reached external memory.

7.5.2 Memory ordering

The **sync** operation should be used:

- to ensure that all uncached writes have completed,
- to enforce a particular ordering of memory operations,
- to flush the write buffer of pending external memory writes.

7.5.3 Coherency between I-side and D-side

There is no coherency guaranteed between the external memory and the D-side and I-side memory subsystems. If coherency is desired then the memory subsystem has to be purged and synchronized.

This is achieved by the following sequence:

1. Flush the entire data cache by issuing **prgset** operations for every set of lines in the cache.
2. Sync the D-side by issuing a **sync.**
3. Invalidate the entire instruction cache by issuing a **prgins.**
4. Sync the I-side by issuing a **syncins.**

This sequence can be modified to perform coherency on a smaller memory region.

7.5.4 Changing memory to uncacheable

Data memory can be cacheable or uncacheable. Instruction memory is always cacheable.

To change an area of data memory from being cacheable to uncachable it is necessary to:

1. Mark region uncacheable in the DPU.
2. Flush it by issuing **prgadd** or **prgset** operations.
3. Sync the data cache by issuing a **sync**.

The region is now uncached and all **loads** and **store** operations to data within it are guaranteed to be uncacheable.

Note: Once a region is marked uncacheable no addresses from that region can become cached; a ***load*** *to an uncacheable address which misses the cache will not cause the address to be allocated in the cache.*

7.5.5 Reset state

After reset all lines in the instruction cache and data cache are marked as invalid. The write buffer and prefetch cache entries are marked as empty.

7.5.6 Cached data in uncached region

Non-cacheable accesses also go to the memory system via the data cache tag ram matching hardware. A side-effect of this implementation is that even non-cacheable accesses will cause a hit if the requested address is already in the cache. This can happen if the user changes an already cached location from cacheable to non-cacheable status.

If data from an uncached region is left in the cache, due to a change in the region's cacheability, then any accesses to this data will be treated as cached.

To prevent this seemingly unpredictable behavior a region that is changed from cacheable to uncachable must be purged from the cache. To change a region to uncached see *Section 7.5.4: Changing memory to uncacheable on page 55*.

7.5.7 Prefetch performance

The prefetch cache is intended to improve performance. This is achieved by explicitly fetching data which lies in external memory and hiding the latency associated with that fetch. A number of points must be borne in mind to make the prefetch cache work effectively.

1 Data must be prefetched well in advance of use. The latency of an external memory access needs to be hidden between the **pft** operation and the first **load** operation which uses data from the prefetched line. This latency is in the region of 30 - 50 bundles for stall free bundles.

2 The prefetch cache size should be taken into account, such that the number of outstanding prefetches does not exceed the number of entries in the prefetch cache.

3 Unused prefetches increase bandwidth and waste entries in the prefetch cache.

The first two points indicate a window for which prefetches might be considered.

8 Streaming data interface (SDI)

8.1 Overview

The ST220 SDI is designed to allow fast and easy connection of on-chip peripherals. Each SDI is unidirectional and includes handshakes to prevent data loss and improve data flow.

The ST220 implements four SDI interfaces, two input ports and two output ports.

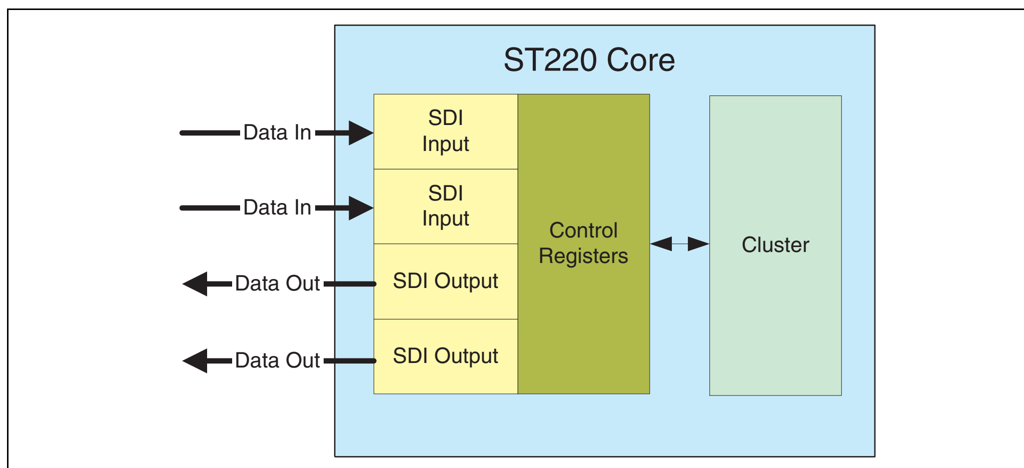


Figure 5: SDI overview

The SDIs:

- provide a mechanism for attaching streaming hardware to the processor core
- reduce STBus traffic and associated processor stall cycles
- reduce cache pollution and control complexity
- prevent deadlock through a timeout mechanism
- allow communication between clock domains without complex synchronization hardware

The SDI ports are accessed via control registers in the core.

8.2 Functional description

Data is communicated synchronously through either output port or input port to the processor. Data is communicated in order, that is, the **n-th** data item communicated will arrive after the **(n - 1)th** item and before the **(n + 1)th** data item.

Writes (**stores**) to an output port will block if the SDI is full. Conversely reads (**loads**) from an input port will block if the channel is empty. The ST220 blocks execution by stalling the entire processor. No execution proceeds until the channel becomes ready for the requested communication, or an interrupt or timeout exception occurs.

Interrupts can also be taken while waiting on the SDI.

8.2.1 Data width

The SDI interface is 32-bits wide.

External to the SDI port, however, the data width can be arbitrary. For example connecting to a DCT peripheral which consumes 16-bits, data could be sent from the ST220 as single 16-bit items. The ST220 can only write 32-bit data to control registers, so writing a pair of 16 bit values in a packed word would be twice as fast. Note in this case the peripheral has to expect pairs of 16-bit values.

8.3 Communication channel

In its basic form the SDI can be used as a communication channel to and from the processor. It is fully synchronized, allowing idealized input and output to be dealt with directly by the processor using **load** and **store** operations directly access the processor registers.

The SDI accesses can be initiated from C program code as accesses to volatile variables.

8.3.1 Timeouts

The timeouts operate as monitors to each individual SDI access. If an access remains stalled for too long, as defined by the control registers, an exception will occur.

8.4 Registers

The SDI interfaces directly to the ST220 load store unit. The interface is through a number of memory mapped registers in the control register address space.

The addresses of these registers are detailed in *Chapter 9: Control registers on page 65*.

8.4.1 Input channel memory mapping

SDI*i*_DATA The SDI*i*_DATA register is the location from which data is read from the input channel. The processor control and channel logic synchronize to ensure no data is lost. If the SDI*i*_DATA register is empty the processor will stall. Writing this register has no effect and the processor will not stall.

SDI*i*_READY The SDI*i*_READY register is implementation specific. If non zero it indicates that the channel has data ready to be read.

This value indicates a minimum number of ready items. Returning the exact amount of data ready to be read from the channel may not be possible for a number of reasons, that is, clock boundary issues, propagation delays, hence the looser condition of the minimum number of ready items. In its simplest form this ready value can be 1, indicating at least one item is ready.

SDI*i*_CONTROL The SDI*i*_CONTROL register is used to reset the channel and the SDI*i*_TIMEOUT register. The usage of the bits are defined in *Table 9*. The definition of the privilege bits is given in *Section 8.4.3: Protection on page 61*.

Name	Bit(s)	Access (U/S)	Reset	Comment
PRIV	[1:0]	RO/RW	0x0	Privilege bits.
RESETINPUT	2	RO/RO	0x0	RESETINPUT (Read Only) acts as RESETREQUEST when slave, RESETACK when master.
RESETOUTPUT	3	RO/RW	0x0	RESETOUTPUT, acts as RESETREQUEST when master, RESETACK when slave.
INPUTNOTOUTPUT	4	RO/RO	0x0	INPUTNOTOUTPUT (Read only).
Reserved	5	RO/RO	0x0	Reserved
MASTERNOTSLAVE	6	RO/RO	0x0	MASTERNOTSLAVE (Read only).
TIMEOUTENABLE	7	RO/RW	0x0	Timeout Disable (set to 1 to disable timeout interrupts).
Reserved	[31:8]	RO/RO	0x0	Reserved

Table 9: SDI0_CONTROL bit fields

SDI*i*_COUNT The SDI*i*_TIMEOUT register is reset to this value each time an SDI data value is successfully accessed. The value may be read or written. At reset it is set to a fixed value defined by the particular implementation. Time-outs can be disabled via the SDI*i*_CONTROL register.

SDI*i*_TIMEOUT The number of cycles an SDI data access will be allowed to stall before a timeout exception will be raised. The value may be read or written. This register will normally be set to the value of SDI*i*_COUNT. Exceptions to this are when it has been specifically set to another value, or when an SDI access has taken a timeout exception or been interrupted. In the case of an SDI timeout the SDI*i*_TIMEOUT register will contain the value zero.

8.4.2 Output channel memory mapping

SDI*i*_DATA The SDI*i*_DATA register is the location from which data is written to the output channel. The processor control and channel logic synchronize to ensure no data is overwritten. If the SDI*i*_DATA register is full, the processor will stall. Reading this value has no effect and the processor will not stall. The value returned is implementation specific.

SDI*i*_READY The SDI*i*_READY register is implementation specific. If non zero it indicates that the channel has space where data can be written.

This value indicates a minimum number of empty spaces where data can be written. In an implementation where the channel is connected to a FIFO this register could indicate, full, notfull, the FIFO is half empty by returning (for example) the values 0, 1, 32. Returning the exact amount of data space available in the channel may not be possible for a number of reasons, that is, clock boundary issues, propagation delays, hence the looser condition of the minimum number of ready items. In the simplest form this ready value can be 1, indicating at least one item can be written.

SDI*i*_CONTROL Bits defined as *Section 8.4.1: Input channel memory mapping on page 59*.

SDI*i*_COUNT Defined as *Section 8.4.1: Input channel memory mapping on page 59*.

SDI*i*_TIMEOUT Defined as *Section 8.4.1: Input channel memory mapping on page 59*.

8.4.3 Protection

The SDI register space is protected from malicious usage via access permissions held in each SDI*i*_CONTROL register. The reset behaviour is that accesses to the SDI registers are only allowed in **supervisor** mode.

The protection can be loosened to allow **user** access to an SDI*i*_DATA and SDI*i*_READY registers. This is achieved via the SDI*i*_CONTROL register, PRIV[1:0] two bit field, indicating the access allowed for each SDI.

SDI_Access_Priv	Allowed privilege	Comment
00	Supervisor User	Any access allowed Protected, will cause exception
01	Supervisor User	Any access allowed Allow access to data and ready register
10	Not defined	Reserved
11	Not defined	Reserved

Table 10: Access Privileges for SDI Ports

8.5 Exceptions, interrupts, reset and restart

8.5.1 Interrupts

While stalled accessing the SDI, the processor can take any processor interrupt. The interrupt will be taken as if it occurred just prior to the bundle accessing the SDI.

Return from interrupt

The **rfi** from the exception handler will continue, as is normal operation, at the point prior to the interrupt.

If the SDI has become ready during execution of the interrupt handler the SDI***i***_TIMEOUT register will be reset to the value in SDI***i***_COUNT.

If however the SDI is not ready, the processor will revert to the stalled state. This will be waiting for the channel to become ready while counting down the SDI***i***_TIMEOUT register from the value held prior to the exception.

Access to SDI registers

The interrupt handler can access all the SDI registers.

Note: Accessing the SDI***i***_DATA *register may alter the state of the processor observed by the interrupted processor.*

8.5.2 SDI exceptions

In this case the EXCEPT_CAUSE register will indicate an SDI timeout exception. The exception address will point to the SDI register on which the processor was waiting when the exception occurred.

An ST220 communications SDI timeout exception will occur if the processor is actively waiting for a response from the interface for longer than the interface's timeout period.

The ST220 exception handler, in the case of a SDI timeout exception, is able to restart the communicating process. This is achieved by executing an **rfi** to the instruction that caused the exception. This will cause re-execution of the instruction accessing the SDI.

Note: The SDI***i***_TIMEOUT *register will need increasing from the zero value that caused the exception, otherwise the exception will be triggered again immediately.*

The timeout exception is generated by the processor and not the channel.

8.5.3 Restart (or soft reset)

A channel can only be restarted by the master of the channel.

A master may be either an input or an output channel. *Figure 6: Soft reset control structure on page 64* shows how the reset structure is connected.

A reset is initiated by the master by setting the RESETREQUEST bit in the CONTROL register. This causes the channel to begin the reset process. Once acknowledged (i.e. RESETACK = 1) as having been received by the slave, and consequently the entire channel structure being reset, the reset is removed (that is, RESETREQUEST = 0) is communicated to the slave port. Once acknowledged (that is, RESTACK = 0), this indicates that the entire channel has exited the reset state.

The slave reset can be used to reset a slave subsystem.

Normally the output channel will be the master. However in cases where the output channel is connected to a dump peripheral it may be necessary to make the input channel the master, particularly where this is a processor interface.

The restart structure outlined will work across asynchronous clock boundaries.

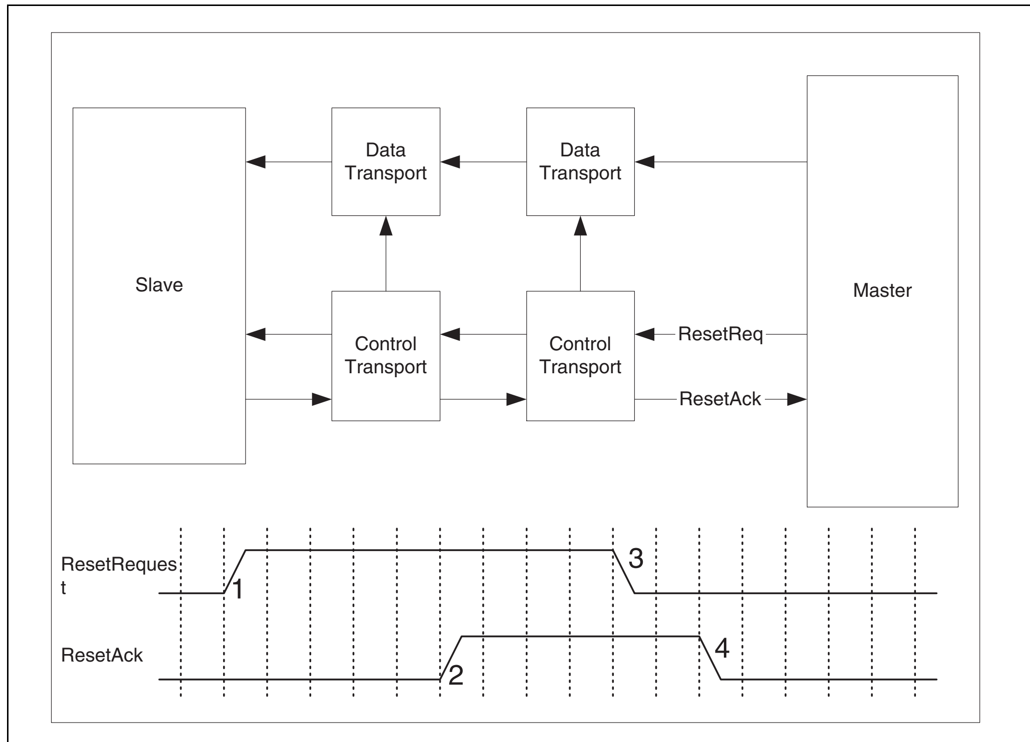


Figure 6: Soft reset control structure

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 subsytems, RESETACK is forwarded to master. System can restart.

9 Control registers

The ST220 control registers contain processor state which is not commonly accessed by application code. This includes accessing the protection units, PSW, exception registers and breakpoint registers.

9.1 Access operations

Control registers[1] are mapped into the address space, allowing access through normal **load** and **store** operations.

All control register accesses are word (32-bit) operations. Byte and half word **load** and **stores** to control registers are not supported and will generate CREG_ACCESS_VIOLATION exceptions.

Dismissible **loads** to control register space always return zero. Control register **loads** or **stores** are executed within the LSU without reference to the DPU regions.

9.2 Exceptions

The control register unit generates an exception when a **load** or **store** tries to:

- access a control register that does not exist (CREG_NO_MAPPING),
- write to a control register without correct permissions (CREG_ACCESS_VIOLATION),

1. Control registers can not be accessed via the STBus.

- perform a byte and half word accesses to control registers (CREG_ACCESS_VIOLATION),
- perform a misaligned word access to a control register (CREG_NO_MAPPING)

For details of the exception cause register see *Section 5.6: Exception types and priorities on page 34*.

9.3 Control register addresses

Below is a table of the control register addresses on the ST220.

The control registers start from a base of 0xFFFF0000, offsets listed in the table are relative to this.

The Access column shows the access rights in **user** and **supervisor** mode.

NA	No access (protection fault)
RO	Read only
RW	Read/Write
CF	Configurable

Name	Offset	Access (U/S)	Comment
PSW	0xfff8	RO/RW	The Program Status Word.
SAVED_PSW	0xfff0	RO/RW	Saved PSW, written by hardware on exception.
SAVED_PC	0xffe8	RO/RW	Saved Program Counter, written by hardware on exception.
HANDLER_PC	0xffe0	RO/RW	The address of the exception handler code.
EXCEPT_CAUSE	0xffd8	RO/RW	A one hot vector of trap (exception/interrupt) types, indicating the cause of the last trap. Written by the hardware on a trap.

Table 11: Control Registers - BASE: CREG_BASE

Name	Offset	Access (U/S)	Comment
EXCEPT_ADDR	0xffd0	RO/RW	This will be the data effective address in the case of either a DPU, CREG, DBREAK, or MISALIGNED_TRAP exception. For other exception types this register will be zero.
ST200_VERSION	0xffc8	RO/RO	The version number of the core. For the ST220 returns 0x01.
SAVED_SAVED_PSW	0xffc0	RO/RW	PSW saved by debug unit interrupt.
SAVED_SAVED_PC	0xffb8	RO/RW	PC saved by debug unit interrupt.
PERIPHERAL_BASE	0xffb0	RO/RO	Base address of peripheral registers. The top 12 bits of this register are wired to the peripheral base input pins.
SCRATCH1	0xffa8	RO/RW	Scratch register reserved for use by the interrupt handler.
SCRATCH2	0xffa0	RO/RW	Scratch register reserved for use by the debug interrupt handler.
DPU_REGION0	0xff80	RO/RW	DPU region address.
DPU_ATR0	0xff78	RO/RW	DPU region attribute.
DPU_REGION1	0xff70	RO/RW	DPU region address.
DPU_ATR1	0xff68	RO/RW	DPU region attribute.
DPU_REGION2	0xff60	RO/RW	DPU region address.
DPU_ATR2	0xff58	RO/RW	DPU region attribute.
DPU_REGION3	0xff50	RO/RW	DPU region address.
DPU_ATR3	0xff48	RO/RW	DPU region attribute.
DPU_REGION4	0xff40	RO/RW	DPU region address.
DPU_ATR4	0xff38	RO/RW	DPU region attribute.
DPU_REGION5	0xff30	RO/RW	DPU region address.
DPU_ATR5	0xff28	RO/RW	DPU region attribute.

Table 11: Control Registers - BASE: CREG_BASE

Name	Offset	Access (U/S)	Comment
DPU_REGION6	0xff20	RO/RW	DPU region address.
DPU_ATR6	0xff18	RO/RW	DPU region attribute.
DPU_REGION7	0xff10	RO/RW	DPU region address.
DPU_ATR7	0xff08	RO/RW	DPU region attribute.
DBREAK_LOWER	0xfe80	RO/RW	Data breakpoint lower address.
DBREAK_UPPER	0xfe78	RO/RW	Data breakpoint upper address.
DBREAK_CONTROL	0xfe70	RO/RW	Data breakpoint control.
IPU_REGION0	0xfe40	RO/RW	IPU region address.
IPU_ATR0	0xfe38	RO/RW	IPU region attribute.
IPU_REGION1	0xfe30	RO/RW	IPU region address.
IPU_ATR1	0xfe28	RO/RW	IPU region attribute.
IPU_REGION2	0xfe20	RO/RW	IPU region address.
IPU_ATR2	0xfe18	RO/RW	IPU region attribute.
IPU_REGION3	0xfe10	RO/RW	IPU region address.
IPU_ATR3	0xfe08	RO/RW	IPU region attribute.
IBREAK_LOWER	0xfdd0	RO/RW	Instruction breakpoint lower address.
IBREAK_UPPER	0xfdc8	RO/RW	Instruction breakpoint upper address.
IBREAK_CONTROL	0xfdc0	RO/RW	Instruction breakpoint control.
PM_CR	0xf800	RO/RW	Performance monitoring control.
PM_CNT0	0xf808	RO/RW	Performance monitor counter 0 value.
PM_CNT1	0xf810	RO/RW	Performance monitor counter 1 value.
PM_CNT2	0xf818	RO/RW	Performance monitor counter 2 value.
PM_CNT3	0xf820	RO/RW	Performance monitor counter 3 value.
PM_PCLK	0xf828	RO/RO	Performance monitor core cycle counter.

Table 11: Control Registers - BASE: CREG_BASE

Name	Offset	Access (U/S)	Comment
SDI0_DATA	0xe000	CF/RW	SDI 0 data.
SDI0_READY	0xe008	CF/RW	SDI 0 ready.
SDI0_CONTROL	0xe010	RO/RW	SDI 0 control.
SDI0_COUNT	0xe018	RO/RW	SDI 0 count.
SDI0_TIMEOUT	0xe020	RO/RW	SDI 0 timeout.
SDI1_DATA	0xe400	CF/RW	SDI 1 data.
SDI1_READY	0xe408	CF/RW	SDI 1 ready.
SDI1_CONTROL	0xe410	RO/RW	SDI 1 control.
SDI1_COUNT	0xe418	RO/RW	SDI 1 count.
SDI1_TIMEOUT	0xe420	RO/RW	SDI 1 timeout.
SDI2_DATA	0xe800	CF/RW	SDI 2 data.
SDI2_READY	0xe808	CF/RW	SDI 2 ready.
SDI2_CONTROL	0xe810	RO/RW	SDI 2 control.
SDI2_COUNT	0xe818	RO/RW	SDI 2 count.
SDI2_TIMEOUT	0xe820	RO/RW	SDI 2 timeout.
SDI3_DATA	0xec00	CF/RW	SDI 3 data.
SDI3_READY	0xec08	CF/RW	SDI 3 ready.
SDI3_CONTROL	0xec10	RO/RW	SDI 3 control.
SDI3_COUNT	0xec18	RO/RW	SDI 3 count.
SDI3_TIMEOUT	0xec20	RO/RW	SDI 3 timeout.

Table 11: Control Registers - BASE: CREG_BASE

Timers

10

Three timers are provided on the ST220. These are controlled by registers mapped into the ST220 memory space (see *Chapter 11: Peripheral addresses on page 75*).

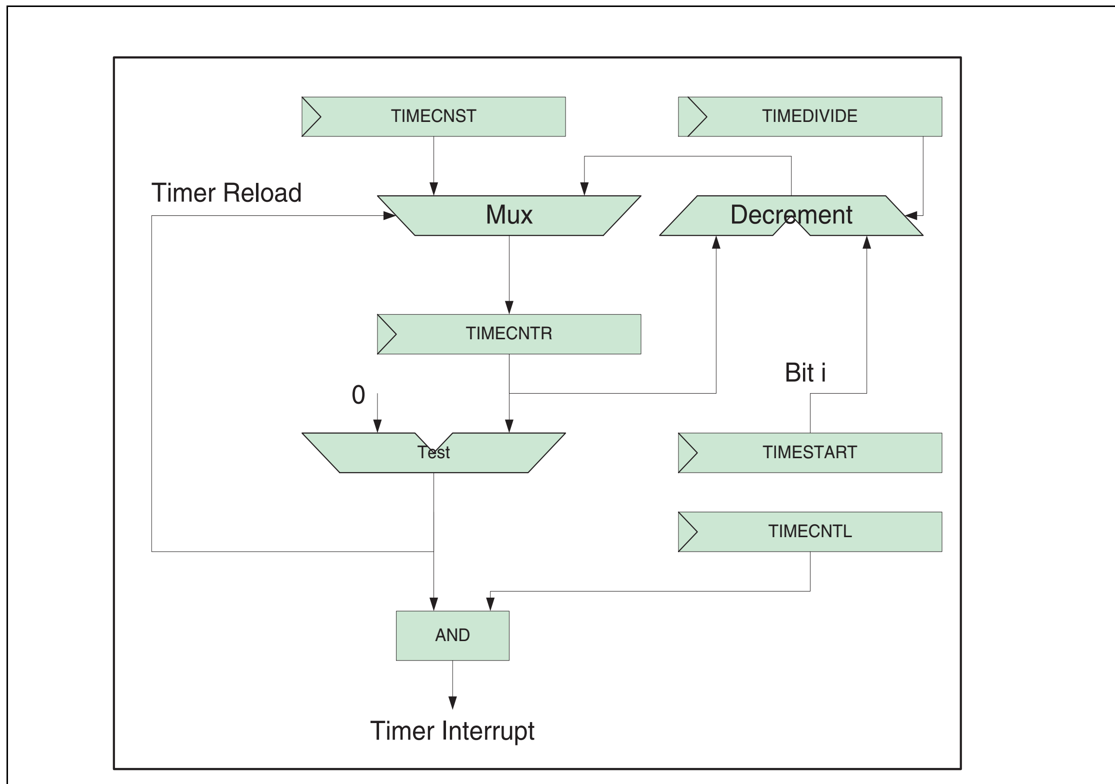


Figure 7: Timers

10.1 Operation

For each of the three timers (***i*** = 0, 1, 2), the TIMECNTR***i*** register is the current value of the timer. This value is decremented on each timer tick until zero is reached. Upon the next tick, TIMECNTR***i*** is loaded with TIMECNST***i*** and the ***i***th timer interrupt is raised if enabled. The TIMECNTL***i*** register controls the enabling of interrupts for each timer.

Timer counting is enabled by the ***i***th LSB of the TIMESTART register. Counters are not reset when disabled. Hence initial values can be written using the TIMECNTR***i*** registers.

The frequency of timer ticks is controlled by programming the TIMEDIVIDE register.

These registers are covered in more detail in the following subsections.

10.1.1 TIMEDIVIDE

The TIMEDIVIDE register sets the number of bus clock cycles between each timer tick. This register can be programmed with values between 0 and 65535 (only the bottom 16 bits are used). The divide value is equal to the value of this register plus one. This register will reset to zero (divide by 1). Writing this register sets the divide value and reading it returns the current divide value.

It is expected that the boot code will setup the TIMEDIVIDE register so that timer ticks occur every 1us.

10.1.2 TIMECNTR*i*

Name	Bit(s)	Access (U/S)	Reset	Comment
COUNT	[31:0]	RW	0x0	Current value of the timer counters.

Table 12: TIMECNTR0 bit fields

The TIMECNTR***i*** registers hold the current values of the timer counters.

Writing to these registers can be used to set initial values for the counters.

10.1.3 TIMECNST*i*

Name	Bit(s)	Access (U/S)	Reset	Comment
CONST	[31:0]	RW	0x0	Constant to be reloaded when the timer reaches zero.

Table 13: TIMECNST0 bit fields

The TIMECNST*i* registers set the value to be reloaded into each corresponding timer on reaching zero. If interrupts are enabled, this will define the number of ticks between interrupts.

10.1.4 TIMECNTL*i*

Name	Bit(s)	Access (U/S)	Reset	Comment
ENABLE	0	RW	0x0	Enable the timer interrupt.
Reserved	[31:1]	RO	0x0	Reserved

Table 14: TIMECNTL0 bit fields

The TIMECNTL*i* registers enable the timer interrupts the processor. The LSB of each register (ENABLE) enables the corresponding timer interrupt.

10.1.5 TIMESTART

Name	Bit(s)	Access (U/S)	Reset	Comment
T0	0	RW	0x0	Enable for timer 0.
T1	1	RW	0x0	Enable for timer 1.
T2	2	RW	0x0	Enable for timer 2.
Reserved	[31:3]	RO	0x0	Reserved

Table 15: TIMESTART bit fields

The TIMESTART register contains enable bits for each timers, (T0 for timer 0, T1 for timer 1 and T2 for timer 2), a set bit indicates enable and a clear bit disable. When disabled, a timer value remains constant.

10.2 Timer interrupts

Timer interrupts use bits 2:0 of the Interrupt Test Register, *Section 12.3: Interrupt registers on page 80*.

Note: These bits remain set when the interrupt is taken and must be explicitly cleared.

10.3 Programming the timer

The TIMECNST*i* registers set the value to be reloaded into the corresponding timer. The value is loaded on the timer tick after a zero is reached, such that, the duration between timers reaching zero is (TIMECNST*i* + 1). (For example, setting it to 99 will cause a reload and timer interrupt (if enabled) every 100 ticks.

11 Peripheral addresses

On the ST220 the interrupt controller, DSU, DSU ROM and the timers are memory mapped peripherals. Under normal usage these peripherals should, with the exception of the DSU ROM, be mapped in an uncacheable region in the DPU.

11.1 Peripheral addresses

Below is a table of the peripheral register addresses on the ST220.

The peripheral registers start from the peripheral base address which can be found by reading the PERIPHERAL_BASE register (see *Chapter 9: Control registers on page 65*).

The Access column shows the access rights:

RO Read only

RW Read/write

CF Configurable

11.1.1 Interrupt controller & timer registers

The interrupt controller and timer registers start from PERIPHERAL_BASE + 0x0.

Name	Offset	Access (U/S)	Comment
INTPENDING0	0x0	RO	Interrupt pending bits 31:0.
INTPENDING1	0x8	RO	Interrupt pending bits 63:32.
INTMASK0	0x10	RW	Interrupt mask bits 31:0.
INTMASK1	0x18	RW	Interrupt mask bits 63:32.
INTTEST0	0x20	RW	Interrupt test register bits 31:0.
INTTEST1	0x28	RW	Interrupt test register bits 63:32.
INTCLR0	0x30	RW	Interrupt clear register bits 31:0.
INTCLR1	0x38	RW	Interrupt clear register bits 63:32.
INTSET0	0x40	RW	Interrupt set register bits 31:0.
INTSET1	0x48	RW	Interrupt clear register bits 63:32.
TIMESTART	0x50	RW	Timer start.
TIMECNST0	0x58	RW	Timer constant.
TIMECNTR0	0x60	RW	Timer counter.
TIMECNTL0	0x68	RW	Timer control.
TIMECNST1	0x70	RW	Timer constant.
TIMECNTR1	0x78	RW	Timer counter.
TIMECNTL1	0x80	RW	Timer control.
TIMECNST2	0x88	RW	Timer constant.
TIMECNTR2	0x90	RW	Timer counter.
TIMECNTL2	0x98	RW	Timer control.
TIMEDIVIDE	0x100	RW	Timer divide.
INTMASKCLR0	0x108	RW	Interrupt mask clear bits 31:0.
INTMASKCLR1	0x110	RW	Interrupt mask clear bits 63:32.
INTMASKSET0	0x118	RW	Interrupt mask set bits 31:0.

Table 16: Interrupt Controller - BASE: INTCR_BASE

Name	Offset	Access (U/S)	Comment
INTMASKSET1	0x120	RW	Interrupt mask set bits 63:32.

Table 16: Interrupt Controller - BASE: INTCR_BASE

11.1.2 DSU registers

The interrupt controller registers start from PERIPHERAL_BASE + 0x3000

Name	Offset	Access (U/S)	Comment
DSR0	0x0	RO	DSU version.
DSR1	0x8	RW	DSU status.
DSR2	0x10	RW	DSU output.
DSR3	0x18	RW	DSU communication.
DSR4	0x20	RW	DSU communication.
DSR5	0x28	RW	DSU communication.
DSR6	0x30	RW	DSU communication.
DSR7	0x38	RW	DSU communication.
DSR8	0x40	RW	DSU communication.
DSR9	0x48	RW	DSU communication.
DSR10	0x50	RW	DSU communication.
DSR11	0x58	RW	DSU communication.
DSR12	0x60	RW	DSU communication.
DSR13	0x68	RW	DSU communication.
DSR14	0x70	RW	DSU communication.
DSR15	0x78	RW	DSU communication.
DSR16	0x80	RW	DSU communication.
DSR17	0x88	RW	DSU communication.
DSR18	0x90	RW	DSU communication.

Table 17: Debug Support Unit - BASE: DSU_BASE

Name	Offset	Access (U/S)	Comment
DSR19	0x98	RW	DSU communication.
DSR20	0xa0	RW	DSU communication.
DSR21	0xa8	RW	DSU communication.
DSR22	0xb0	RW	DSU communication.
DSR23	0xb8	RW	DSU communication.
DSR24	0xc0	RW	DSU communication.
DSR25	0xc8	RW	DSU communication.
DSR26	0xd0	RW	DSU communication.
DSR27	0xd8	RW	DSU communication.
DSR28	0xe0	RW	DSU communication.
DSR29	0xe8	RW	DSU communication.
DSR30	0xf0	RW	DSU communication.
DSR31	0xf8	RW	DSU communication.

Table 17: Debug Support Unit - BASE: DSU_BASE

11.1.3 DSU ROM

The DSU ROM starts from PERIPHERAL_BASE + 0x4000 (see *Chapter 13: Debugging support on page 85*).

12 Interrupt controller

The ST220 interrupt controller supports up to 64 interrupt sources. The system is programmed via a number of 64-bit memory-mapped control registers.

12.1 Architecture

The structure of the interrupt controller is shown in *Figure 8*.

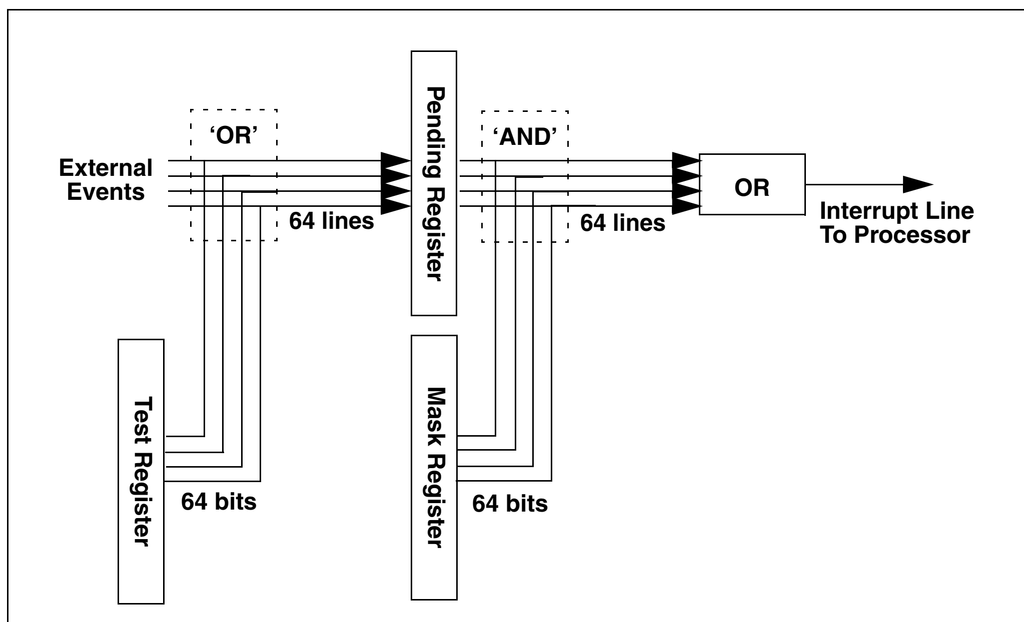


Figure 8: Interrupt controller

12.2 Operation

An interrupting event takes an interrupt line high. This is then sampled, causing the corresponding bit in the 64-bit INTPENDING register to be set. The INTPENDING register is then parallel AND-ed with the INTMASK register. The masked interrupts are then OR-red into a single interrupt line that is presented to the processor core.

This architecture ensures that:

- all external interrupts will interrupt the processor,
- interrupts can be individually enabled or disabled.

Setting or clearing bits in the INTMASK register enables or disables the corresponding interrupt lines.

12.2.1 Test register

External interrupts are wide OR-ed with the contents of the INTTEST register before being sampled by the INTPENDING register. This allows the programmer to simulate interrupts into the processor for test purposes.

12.3 Interrupt registers

For the addresses of these memory mapped registers see *Chapter 11: Peripheral addresses on page 75*.

12.3.1 Interrupt pending register

The 64-bit INTPENDING register holds the current interrupt status. Bits in this register are set by external interrupts or the INTTEST register.

A number of bits in the INTPENDING register are preassigned to ST220 peripherals. The remaining bits can be assigned to other peripherals or external devices.

Name: INTPENDING0[31:0], INTPENDING1[63:32]

Name	Bit(s)	Access (U/S)	Reset	Comment
TIMER0	0	RO/RO	0x0	Interrupt is pending from timer 0.
TIMER1	1	RO/RO	0x0	Interrupt is pending from timer 1.
TIMER2	2	RO/RO	0x0	Interrupt is pending from timer 2.
Reserved	[15:3]	RO/RO	0x0	System defined - refer to data sheet.
Reserved	[31:16]	RO/RO	0x0	System defined non-maskable interrupts - refer to data sheet.

Table 18: INTPENDING0 bit fields

Name	Bit(s)	Access (U/S)	Reset	Comment
Reserved	[31:0]	RO/RO	0x0	System defined - refer to data sheet.

Table 19: INTPENDING1 bit fields

12.3.2 Interrupt mask register (INTMASK)

The INTMASK register is a 64 bit register whose contents are AND-ed with the INTPENDING register. It is used to enable and disable external interrupts.

Name: INTMASK0[31:0], INTMASK1[63:32]

Reset Value: 0

Interrupts are enabled by setting, and disabled by clearing, the corresponding bits the INTMASK register.

Bits 31:16 are fixed set, so that the corresponding interrupt lines are permanently enabled.

INTMASKCLR and INTMASKSET

Two 64-bit address locations in the interrupt controller memory space support bit-wise access to the INTMASK register. A store to these locations causes the corresponding bits in the INTMASK register to be cleared or set.

Name: INTMASKCLR0[31:0], INTMASKCLR1[63:32], INTMASKSET0[31:0], INTMASKSET1[63:32]

Reset Value: 0

Using this method of accessing the INTMASK register avoids any problems caused by interrupts occuring during a Read-Modify-Write sequence and therefore avoids the need to have interrupts disabled while modifying these registers.

12.3.3 Interrupt test register (INTTEST)

The INTTEST register is a 64-bit register whose contents are OR-ed with external interrupts. It provides a mechanism for simulating interrupts to the processor.

Name: INTTEST0[31:0], INTTEST1[63:32]

Reset Value: 0

Setting bits in the INTTEST register causes the corresponding bits in the INTPENDING register to be set.

INTCLR and INTSET

Two 64-bit address locations in the interrupt controller memory space support bit-wise access to the INTTEST register. A store to these locations causes the corresponding bits in the INTTEST register to be cleared or set.

Name: INTCLR0[31:0], INTCLR1[63:32], INTSET0[31:0], INTSET1[63:32]

Reset Value: 0

Using this method of accessing the INTTEST register avoids the overhead in a direct write of having to read, save and restore unaccessed bits.

12.4 Programming

12.4.1 Enabling/disabling interrupts

Interrupts are enabled and disabled by setting and clearing the appropriate bits in the INTMASK register.

In the ST220 interrupts 31:16 are non-maskable and permanently enabled. INTMASK register bits 31:16 are tied to high.

The INTMASK register can written to directly as two 32-bit words, or bit-wise using the mask clear and mask set locations, see *INTMASKCLR and INTMASKSET on page 82*.

12.4.2 Test register

Interrupts can be simulated for test purposes by setting bits in the INTTEST register. This register is directly OR-red with the external interrupt signals into the INTPENDING register.

The INTTEST register can written to directly as two 32-bit words, or bit-wise using the test clear and test set locations, see *INTCLR and INTSET on page 82*.

Note: INTTEST *register bits are not reset when the interrupt is taken and must be explicitly cleared by the program.*

12.4.3 Interrupt priority

The interrupt handling code is responsible for prioritization of interrupts. No hardware support is provided.

12.4.4 Timer interrupts

The first three bits in the INTTEST register are also used in conjunction with the timer. When a timer interrupt occurs the corrisponding bit in the INTTEST register is set (timer 0 sets bit 0, timer 1 sets bit 1 and timer 2 sets bit 2). This prevents timer interrupts from being lost. When a timer interrupt is serviced the software should clear the correct bit in the INTTEST register.

13 Debugging support

13.1 Overview

Debugging support on the ST220 is provided by 4 main components.

- Core

  The ST220 core includes a non maskable debug interrupt, and additional state to support the taking of debug interrupts. The core also contains hardware breakpoint support.

- DSU

  Shared DSU registers and state machine which generates debug interrupts and send responses over debug interface.

- Debug ROM

  Default program run in response to debug interrupt. This program uses the DSU registers to send higher level protocols over the debug interface. This program implements the **DSU_PEEK**, **DSU_POKE**, **DSU_CALL_OR_RETURN** and **DSU_FLUSH** operations.

- Host debug interface

  The hardware link, using the TAPlink protocol, to any connected host target interface (HTI). Supports **peek**, **poke**, **peeked** and **event** messages.

13.2 Core

13.2.1 Debug interrupts

The ST220 can accept and service interrupts from the DSU. Debug interrupts are higher priority than normal interrupts, cannot be masked, and place the ST220 in a debug state.

Debug interrupt handling

A debug interrupt is handled differently to other external interrupts.

Taking a debug interrupt can be summarized as:

```
NEXT_PC ← DEBUG_HANDLER_PC;     // Branch to handler

SAVED_SAVED_PSW ← SAVED_PSW;    // Save the SAVED_PSW and
SAVED_SAVED_PC ← SAVED_PC;      // SAVED_PC

SAVED_PSW ← PSW;                // Save the PSW and PC
SAVED_PC ← BUNDLE_PC;           //

PSW[USER_MODE] ← 0;      // Enter supervisor mode
PSW[INT_ENABLE] ← 0;     // Disable interrupts
PSW[IBREAK_ENABLE] ← 0;  // Disable instruction breakpoints
PSW[DBREAK_ENABLE] ← 0;  // Disable data breakpoints
PSW[IPU_ENABLE] ← 0;     // Disable the IPU
PSW[DPU_ENABLE] ← 0;     // Disable the DPU
PSW[DEBUG_MODE] ← 1;     // Enter debug mode
```

Exiting debug mode

Debug mode is exited, and normal mode re-entered, when the DEBUG_MODE bit is cleared in the PSW. The only supported method for writing the PSW is via the **rfi** operation (see *Section 3.3.4: Supported method for changing the PSW*).

Note: Although clearing the DEBUG_MODE bit causes the ST220 to exit debug mode, attempting to set the DEBUG_MODE bit when it is not already set does not cause the core to enter debug mode and the DEBUG_MODE bit remains clear. Debug mode can only be entered by taking a debug interrupt from the DSU.

13.2.2 Hardware breakpoint support

Breakpoints are supported by:

- enable bits in the PSW,
- address registers to define memory ranges,
- control register to specify comparison operations.

The safe way to use the breakpoint registers is to disable the breakpoints and then set the control and address registers before enabling the breakpoints again. This will prevent spurious breaks due to inconsistent control and address registers.

Enable bits

Breakpoints are enabled though the PSW, one bit for instruction breakpoints and another data breakpoints (see *Section 3.3: Program status word (PSW)*).

Address registers

Two 32-bit registers are used to define addresses for the instruction and data breakpoints (IBREAK_LOWER, DBREAK_LOWER, IBREAK_UPPER, DBREAK_UPPER). These registers are all reset to the value 0.

Control registers

Bits [3:0] in this register determine the comparison operations performed on the breakpoint addresses. If the comparison is true then a breakpoint exception (IBREAK or DBREAK) is signaled.

For instruction breakpoints the bundle address is used for comparison. For data breakpoints, the data effective address is used for comparison. The following comparison operations are defined.

Bit	Break point comparison
0	Address <= Upper && Address >= Lower
1	Address > Upper || Address < Lower
2	Address == Upper || Address == Lower
3	Address & Upper == Lower

Table 20: Breakpoint register comparisons

13.3 Debug support unit

The DSU allows both software and hardware to be debugged from a host by giving direct access to the ST220 core.

13.3.1 Architecture

The architecture of the DSU is shown in *Figure 9*.

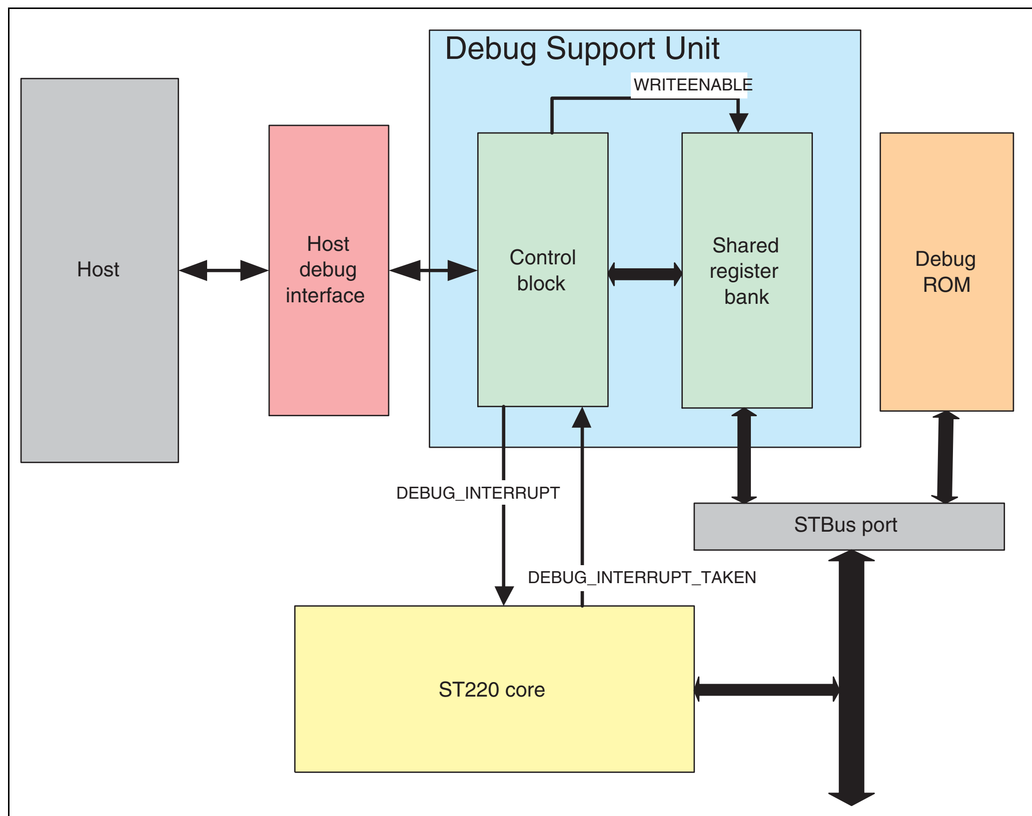


Figure 9: DSU architecture

The DSU is controlled by a host via the debug interface. The DSU control block interacts directly with the ST220 core via the DEBUG_INTERRUPT and DEBUG_INTERRUPT_TAKEN signals, and the shared register block.

The shared registers can also be accessed via the STBus port.

13.3.2 Shared register bank

The 32 shared registers consists of 3 reserved registers (DSR0-2) and 29 general purpose registers (DSR3-31). These are used to implement communication between the host and the target by the debug handler.

The shared register bank is 32-bits wide and only supports 32-bit STBus operations.

DSU shared registers are listed below.

Register Name	Description	RW	Comments
DSR0	DSU version register	R	Contains ID numbers for DSU, core, and chip version.
DSR1	DSU status register	Bits 0-5 R, others RW	Contains DSU control and status bits.
DSR2	DSU output register	RW	Supports message transfer from target to HTI.
DSR3-31		RW	General purpose registers.

The STBus addresses of the DSU registers are detailed in *Chapter 11: Peripheral addresses on page 75*.

Shared access conventions

The DSU shared registers are accesable independently from both the DSU and the STBus. The ST220 and DSU have no hardware support for synchronizing writes, so software conventions are used to prevent write conflicts.

13.3.3 DSU control registers

DSU version register (DSR0)

The DSU version register is a read-only ID register:

Name	Bit(s)	Access (U/S)	Reset	Comment
PRODUCT_ID	[15:0]	RO	0x0002	Chip ID.
CORE_VERSION	[23:16]	RO	0x02	ST220 core version number.
DSU_VERSION	[31:24]	RO	0x02	DSU design version number.

Table 21: DSR0 bit fields

DSU status register (DSR1)

The DSU status register contains the DSU status and control bits.

Name	Bit(s)	Access (U/S)	Reset	Comment
DEBUG_INTERRUPT_TAKEN	0	RO	0x0	Value of DEBUG_INTERRUPT_TAKEN signal, active high.
SUPERVISOR_WRITE_ENABLE	1	RW	0x1	STBus writes enabled if the core is in supervisor mode (regardless of debug mode).
USER_WRITE_ENABLE	2	RW	0x0	STBus writes enabled if the core is in user mode (regardless of debug mode).
Reserved	[4:3]	RO	0x0	Reserved
OUTPUT_PENDING	5	RO	0x0	DSR2 contains a byte to be sent to the HTI which has not yet been sent.
HW_FLAGS	[15:6]	RW	0x0	Reserved for future hardware purposes.
SW_FLAGS	[31:16]	RW	0x0	Reserved for future software use.

Table 22: DSR1 bit fields

DSU output register (DSR2)

The lower 8 bits of the DSU output register are sent to the TAPlink (to any attached host) on being written.

Name	Bit(s)	Access (U/S)	Reset	Comment
DATA	[7:0]	RW	0x0	Output data.
Reserved	[31:8]	RO	0x0	Always zero.

Table 23: DSR2 bit fields

If OUTPUT_PENDING is non-zero then the byte most recently written has not yet been sent to the HTI and additional writes to the DSR2 will not affect the byte being sent even if they change the contents of the register.

Messages sent via the DSR2 may be delayed if the DSU is busy.

13.4 Debug ROM

The 1024-byte debug ROM is an ST220 peripheral. This contains the debug initialization loop and the default debug handler.

13.4.1 Debug initialization loop

On reset the ST220 starts executing at the beginning of the boot ROM. However, if the DEBUG_ENABLE signal is asserted execution starts at the debug initialization loop (this is the first word of the debug ROM). This word contains a single syllable bundle which loops back to the same location, allowing the DSU to intervene and configure the core before it executes any code.

Note: *Where the* DEBUG_ENABLE *signal cannot be asserted, the boot ROM should start with a tight loop, or perhaps just a delay loop, to allow time for the DSU to interrupt the processor before it takes any action.*

13.4.2 Default debug handler

The default debug handler program starts at the second word of the debug ROM. It supports simple host-target debugging and the ability to install a more complex debug handler. The STBus address of the ROM is given in *Chapter 11: Peripheral addresses on page 75*.

Operation

On taking a debug interrupt, the default debug handler is executed. This first tests if a user handler is installed, DSR3 non zero, and if so branches to this address. Otherwise the handler waits in the command loop.

Command loop

The command loop reads and processes commands from a host, delivered via the TAPlink, to the DSU shared registers. Usage of the designated registers is shown in *Table 24*.

Register name	Host use	Target use
DSU_COMMAND	Set with command	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

Table 24: Command register usage

When the command is complete, the default debug handler stores the results in the argument registers and sets a success code in the response register.

Default handler commands

`DSU_PEEK (DSU_COMMAND =4)`

Reads the 32-bit memory location addressed by DSU_ARG1 and returns the data in DSU_ARG1. The address must be word aligned. If the operation is successful DSU_RESPONSE is set to DSU_PEEKED (1) and a TAPLINK_EVENT_DEFAULT (reason=7) event is sent to the HTI.

Note: Any code greater than 4 is interpreted as a DSU_PEEK *command.*

DSU_POKE (DSU_COMMAND = 3)

Writes the 32-bit data word in DSU_ARG2 to the memory location addressed by DSU_ARG1. The address must be word aligned. If the operation is successful DSU_RESPONSE is set to DSU_POKED (2) and a TAPLINK_EVENT_DEFAULT (reason=7) event is sent to the HTI.

DSU_CALL_OR_RETURN (DSU_COMMAND = 1)

Calls the routine addressed by DSU_ARG1. If the called routine does not return this is effectively a branch. If DSU_ARG1 is zero this is a return call. Just before calling the user routine, or returning from a call, DSU_RESPONSE is set to DSU_RETURNING (3) and a TAPLINK_EVENT_DEFAULT (reason=7) event is sent to the host.

DSU_FLUSH (DSU_COMMAND = 2)

Flushes the address range delimited by DSU_ARG1 and DSU_ARG2 from data and instruction caches.

If a command was successful DSU_RESPONSE is set to DSU_FLUSHED (4) and a TAPLINK_EVENT_DEFAULT (reason=7) event is sent to the host.

Trap handler

If a trap occurs while a command is being processed, for example, an invalid address supplied on a **peek** or **poke**:

- the operation in progress is completed by loading the PC of the offending bundle, the exception cause, and the exception address into DSR_ARG1, DSR_ARG2 and DSR_ARG3 respectively.
- DSU_RESPONSE is set to DSU_GOT_EXCEPTION (Code=5) and a TAPLINK_EVENT_DEFAULT (Reason=7) event is sent to the HTI.
- As for all exceptions, the SAVED_PC and SAVED_PSW registers will have been updated when the exception occured.

Context restore

Prior to exit the default handler restores any state it has altered.

Note: The context may have been additionally altered by commands issued.

Default handler register usage

The following DSU registers are defined and used by the default debug handler program:

DSR number	Designation	Comment
DSR3	DSR_USER_DEBUG_HANDLER	Control switches to this address if content is non-zero.
DSR4-8[a]	DSU_ARG4-8	Not used in current debug handler.
DSR9[a]	DSU_ARG3	Command argument 3.
DSR10[a]	DSU_ARG2	Command argument 2. Used by DSU_POKE and DSU_FLUSH.
DSR11[a]	DSU_ARG1	Command argument 1. Used by all DSU commands.
DSR12	DSU_COMMAND	Command register. Written by HTI, cleared by target when command accepted.
DSR13	DSU_RESPONSE	Response register. Set by target to a completion code, cleared by HTI before issuing next command.
DSR14	Context saving	Saves SCR4_REG[b]
DSR15	Context saving	Saves SCR1_REG[b]
DSR16	Context saving	Saves SCR2_REG[b]
DSR17	Context saving	Saves SCR3_REG[b]
DSR18	Context saving	Saves the branch bits
DSR19	Context saving	Saves LINK_REG[b]
DSR20	Context saving	Saves HANDLER_PC
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 EXCAUSE
DSR26	Context saving	Saves EXADDRESS

Table 25: DSU command registers

DSR number	Designation	Comment
DSR27-30	Unused	Unused
DSR31	Context saving	Saves DSR1

Table 25: DSU command registers

a. Argument registers are placed before the command register in the address space so that a command and its arguments can be loaded with a single **poke** operation.

b. As defined by the ABI (see *ST200 Programming Manual*)

13.5 Host debug interface

Exchange of information with the host is via a host target interface (HTI) adaptor. The DSU connects to the HTI via a JTAG interface, and the HTI connects to the host via Ethernet, USB, serial port or parallel port. This is illustrated in *Figure 10*.

All host-target communication is done with **peek**, **poke**, **peeked** and **event** messages sent between the host and the DSU.

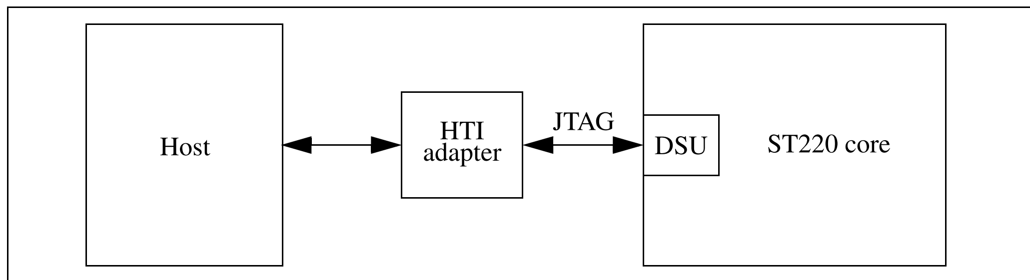


Figure 10: DSU overview

13.5.1 Message format

Commands are sent to the DSU in Taplink message format consisting of a bidirectional byte stream which is interpreted by the DSU as a stream of commands. *Figure 11* shows the DSU commands in Taplink message format.

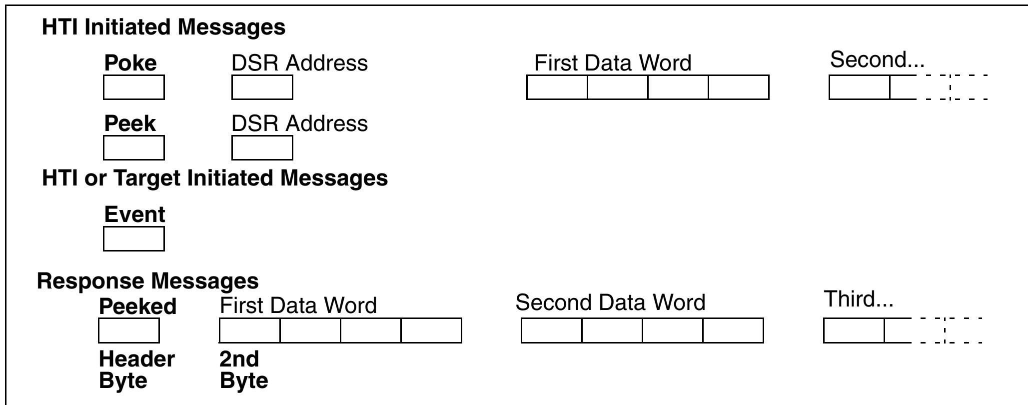


Figure 11: DSU commands

Note: Messages are transmitted little endian, independent of the endianness of the ST220 core.

Header bytes

Header bytes contain command-specific information such as the range of registers to be accessed. Header byte formats for the 4 DSU commands are illustrated in *Figure 12*.

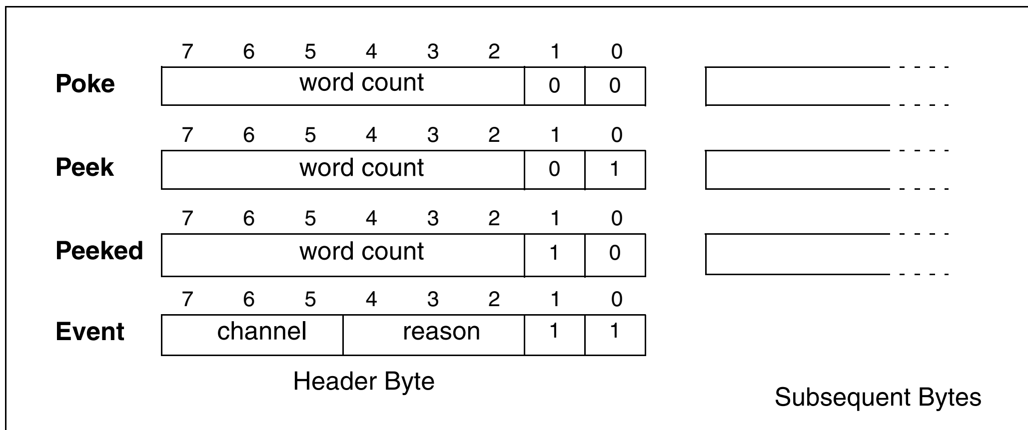


Figure 12: Header bytes

Peek and poke operation

Peek and **poke** commands read and write the shared DSU registers. Each command uses a 6-bit count and one byte as a register address. The byte address references the first register in the range, and the count indicates how many registers are accessed. Counts greater than 32 are interpreted as 32.

Register addresses wrap, so a **peek** or **poke** with an address of 30 and a count of 4 will access registers DSR30, DSR31, DSR0 and DSR1, in that order.

The result of a **peek** command is returned to the host using a **peeked** message.

*Note: **Peeked** messages are not supported from the host to the target and their behavior is undefined.*

13.5.2 Operation

Generating debug interrupts

In order to interrupt the core, the host sends an **event** with reason=1. The **event** is decoded by the DSU and a DEBUG_INTERRUPT signal is sent to the ST220. When the ST220 takes the interrupt (as described in *Debug interrupt handling on page 86*) the DEBUG_INTERRUPT_TAKEN signal goes high.

The functionality available to the host is then dependent on the debug handler program running. The default handler uses designated shared registers to provides the higher level operations detailed in *Default handler commands on page 92*.

Core initiated events

The core can also request service from the host by sending it an **event** message. This is done by writing the **event** to the DSR2 (the output register). The channel and reason fields of the event message are not examined by the hardware and can be used as desired by the software.

14 Performance monitoring

The ST220 provides a hardware instrumentation system which consists of a control register (PM_CR), a core clock counter (PM_PCLK) and four event counters (PM_CNT*i*, *i* = 0, 1, 2, 3). They are all mapped to addresses in the control register space as defined in *Section 9.3: Control register addresses on page 66*.

The system allows the user to simultaneously monitor up to four of the sixteen predefined events.

14.1 Events

The programmable events supported by the ST220 are the following:

Unit	Encoding	Event	Description
Data cache	00000	DHit	Data cache hits. The number of **load** and **stores** that hit the cache. This includes uncached accesses that hit the cache.
	00001	DMiss	The number of **load** and **stores** that miss the cache. This includes **stores** that miss the cache and are sent to the write buffer. Uncached accesses are not included in this count.
	00010	DMissCycles	The number of cycles the core is stalled due to the data cache being busy.

Table 26: Monitored events

Unit	Encoding	Event	Description
Data cache (cont'd)	00011	PftIssued	The number of prefetches that are sent to the STBus.
	00100	PftHits	The number of cached **loads** that hit the prefetch buffer.
	00101	WBHits	The number of writes that hit the write buffer.
Instruction cache	00110	IHit	The number of accesses the instruction buffer made that hit the instruction cache.
	00111	IMiss	The number of accesses the instruction buffer made that missed the instruction cache.
	01000	IMissCycles	The number of cycles the instruction cache was stalled for.
	01001	IBufInvalid	Duration where IBuffer is not able to issue a bundles to the pipeline.
Core	01010	Bundles	Bundles executed
	01011	LDST	Load/Store instructions executed
	01100	TakenBr	Number of taken branches
	01101	NotTakenBr	Number of branches not taken
	01110	Exceptions	Number of exceptions and debug interrupts
	01111	Interrupts	Number of interrupts
Reserved	1xxxx	Undefined	These 16 event numbers are unused on the ST220 and are reserved for future usage.

Table 26: Monitored events

All the events relating to the architectural state of the machine are sampled when bundles commit.

14.2 Access to registers

As all the performance monitoring registers are mapped into the control register space, writing to them is only supported in supervisor mode. An attempt to write to a register in user mode will cause a CREG_ACCESS_VIOLATION exception.

Reading from the registers is permitted in both user and supervisor modes.

14.3 Control register (PM_CR)

The control register is used to reset and enable all the counters, and define the events of the four programmable count registers. The purpose of each of the fields is given below:

Name	Bit(s)	Access (U/S)	Reset	Comment
ENB	0	RO/RW	0x0	When 1, counting is enabled. When 0 counting is disabled.
RST	1	RO/RW	0x0	When a 1 is written all the counters (PM_CNT0-3 and PM_PCLK) are set to zero. If a 0 is written it is ignored. This field does not retain its value and so always reads as 0.
Reserved	[11:2]	RO/RW	0x0	Reserved
EVENT0	[16:12]	RO/RW	0x0	5-bit field specifying the event being monitored for this counter. Only values 0-15 have defined events associated with them.
EVENT1	[21:17]	RO/RW	0x0	5-bit field specifying the event being monitored for this counter. Only values 0-15 have defined events associated with them.
EVENT2	[26:22]	RO/RW	0x0	5-bit field specifying the event being monitored for this counter. Only values 0-15 have defined events associated with them.
EVENT3	[31:27]	RO/RW	0x0	5-bit field specifying the event being monitored for this counter. Only values 0-15 have defined events associated with them.

Table 27: PM_CR bit fields

Writing to the PM_CR updates the value and reading from the PM_CR returns the register's current value.

14.4 Event counters (PM_CNT*i*)

Each of the four event counters is incremented by one each time the countable event specified in the PM_CR occurs. The four programmable event counters can record any one of the events specified in *Table 26 on page 99*.

Reading from these registers return the current event count. Writing will change the current count. If a counter is written at the same time as an event triggers the counter to increment then the increment is ignored.

14.5 Clock counter (PM_PCLK)

The PM_PCLK register is read only. Reading the PM_PCLK register returns a 32-bit value. Writes to PM_PCLK are ignored. This counter will silently wrap back to zero when it overflows.

14.6 Recording events

To start recording, an ST220 general purpose register needs to be written with the desired fields. This can be achieved by first reading the PM_CR register, then modifying it as appropriate. The ENB bit needs to be set to 1. The RST bit needs to be set to 1 if the counters are to be reset. The four programmable counter fields need to be set or modified to reflect change in events being recorded. The value in the register is then written to the memory mapped PM_CR for the operation to begin.

To stop recoding, the value of PM_CR be read, the ENB bit set to zero, and then written back to PM_CR. No other bits must be changed. If the RST bit is set to 1 then the PM_CNT*i* registers will be reset.

Whilst counting events over a long period of time, the 32-bit counters may overflow. This overflow will happen silently and the values will wrap around to zero. To obtain a continuous profile the counters must be read and reset at appropriate regular intervals (depending on the core clock frequency).

Cntr3 Cntr2 Cntr1 Cntr0 0 (Res) r e

Instruction Cache

Data Cache

Core

32:1 Mux

32:1 Mux

32:1 Mux

32:1 Mux

/5 /5 /5 /5 /1 /1

PM_CNT0 (Incrementor 32-bit)

PM_CNT1 (Incrementor 32-bit)

PM_CNT2 (Incrementor 32-bit)

PM_CNT3 (Incrementor 32-bit)

P Clk

PM_PCLK (Incrementor 32-bit)

Figure 13: Performance monitor hardware organization

Execution model

15

15.1 Introduction

This chapter defines the way in which bundles are executed in terms of their component operations.

In the absence of any exceptional behavior the execution is straightforward.

The bundle is fetched from memory. The operations within it are decoded, and their operands read. The operations then execute and writeback their results to the architectural state of the machine. It is important to note that all instructions 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 exceptional behavior the commit point is used to distinguish between recoverable and non-recoverable exceptions.

Exceptions which can be detected prior to the commit point are treated as recoverable. They are recoverable because the machine state has not been updated, hence the state prior to the execution of the bundle can be recovered. In some cases the cause of the exception can be corrected and the bundle restarted.

Conversely non-recoverable exceptions are detected after the commit point. Machine state has been updated and in some cases it may not even be clear which bundle caused the exception. Non-recoverable exceptions are naturally of a serious nature and cannot be restarted. The cause is normally an error in the external memory system, these translate to a bus error exception. On the ST220 this is the only non-recoverable exception.

15.2 Bundle fetch, decode, and execute

The fetching, decoding and executing of bundles is specified 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 all actions associated with one bundle are completed before any actions associated with the next are started.

Implementations will generally make substantial optimizations over this abstract model. However, for typical well-disciplined bundle sequences these effects will not be architecturally visible. A fuller description of the behavior in other cases is defined by the *Chapter 5: Traps: exceptions and interrupts on page 31*.

Note: This simple model does not take into account the latency constraints of operands, and is therefore only valid for hazard free code. All code generated by the compiler is hazard free.

The execution flow shown in *Figure 14* uses notation defined in *Chapter 16: Specification notation on page 109*. Additional functions that have been used to abstract out details are described in *Section 15.3*.

Figure 14: Execution model

15.3 Functions

The flow chart, *Figure 14*, contains a number of functions which abstract out some the details. Those functions are described in this section. Starting with those used in the decode phase, then execution of operations, and finally the exceptional cases.

15.3.1 Bundle decode

Function	Description
NumWords(address)	Returns the number of words in the bundle. 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**.

Table 28: Bundle decode functions

15.3.2 Operation execution

Function	Description
Pre-commit(n)	For the operation **n**th operation in the bundle, execute the Pre-commit phase (*Section 17.3*)[a].
Commit(n)	For the operation **n**th operation in the bundle, execute the Commit phase (*Section 17.3*)[a].

Table 29: Operation execution functions

a. Where **n** is in the range [1 .. number of operations in the bundle] inclusive.

15.3.3 Exceptional cases

Function	Description
InitiateExceptionHandler()	Execute the statements defined in *Section 5.3: Saved execution state on page 32*.
InitiateDebugIntHandler()	Execute the statements defined in *Debug interrupt handling on page 86*.

Table 30: Operation execution functions

16 Specification notation

16.1 Overview

The language used to describe the operations, exceptions and interrupts has the following features:

- a simple variable and type system (see *Section 16.2*),
- expressions (see *Section 16.3*),
- statements (see *Section 16.4*),
- notation for the architectural state of the machine (see *Section 16.5*).

Additional mechanisms are defined to model memory (*Section 16.6.2*), control registers (*Section 16.6.3*), and cache instructions (*Section 16.6.4*).

Each instruction is described using informal text as well as the formal language. Sometimes it is inappropriate for one of these descriptions to convey the full semantics. In such cases the two descriptions must be taken together to constitute the full specification. In the case of an ambiguity or a conflict, the notational language takes precedence over the text.

16.2 Variables and types

Variables are used to hold state. The type of a variable determines the set of values that the variable can take and the available operators. 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 are used to give a bit-accurate representation.

Additional variables are used to hold temporary values. The type of temporary variables is determined by their context rather than explicit declaration. The type of a temporary variable is an integer, a boolean or an array of these.

16.2.1 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 do not 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. These do not change the represented value but can be used as spacers to aid readability.

16.2.2 Boolean

A boolean variable can take two values:

- Boolean false. The literal representation of boolean false is **FALSE**.
- Boolean true. The literal representation of boolean true is **TRUE**.

16.2.3 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 usual way. If bit **i** of a **b**-bit bit-field, where **i** is in **[0, b)**, is set then it contributes $\mathbf{2^i}$ to the integral value of the bit-field. The integral value of the bit-field as a whole is an integer in the range $\mathbf{[0, 2^b)}$.

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 $\mathbf{-2^{(b-1)}}$ to the integral value of the bit-field. The integral value of the bit-field as a whole is an integer in the range $\mathbf{[-2^{b-1}, 2^{b-1})}$.

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.

16.2.4 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 $\mathbf{i}^{th.}$ 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.

16.3 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.

16.3.1 Integer arithmetic operators

Since the notation uses straightforward mathematical integers, the set of standard mathematical operators is available and already defined.

The standard dyadic operators are listed in *Table 31*.

Operation	Description
i + j	Integer addition
i - j	Integer subtraction
i × j	Integer multiplication
i / j	Integer division*
i \ j	Integer remainder*
*	These operators are defined only for **j <> 0**

Table 31: Standard dyadic operators

The standard monadic operators are described in *Table 32*.

Operator	Description
- i	Integer negation
|i|	Integer modulus (absolute value)

Table 32: Standard monadic operators

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 hold where **d≠0**:

$$n = d \times (n / d) + (n \backslash d)$$
$$(-n) / d = -(n / d) = n / (-d)$$
$$(-n) \backslash d = -(n \backslash d)$$
$$n \backslash (-d) = n \backslash d$$
$$0 \leq (n \backslash d) < d \quad \text{where} \quad n \geq 0 \quad \text{and} \quad d > 0$$

16.3.2 Integer shift operators

The available integer shift operators are listed in *Table 33*.

Operation	Description
n << b	Integer left shift
n ≫ b	Integer right shift

Table 33: Shift operators

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 $\mathbf{2^b}$ but with the result rounded towards minus infinity. This contrasts with division, which rounds towards zero, and is the reason why the right shift definition is separate for positive and negative **n**.

16.3.3 Integer bitwise operators

The available integer bitwise operators are listed in *Table 34*.

Operation	Description
$i \wedge j$	Integer bitwise **AND**
$i \vee j$	Integer bitwise **OR**
$i \oplus j$	Integer bitwise **XOR**
$\sim i$	Integer bitwise **NOT**
$n_{\langle b\ FOR\ m \rangle}$	Integer bit-field extraction: extract **m** bits starting at bit **b** from integer **n**
$n_{\langle b \rangle}$	Integer bit-field extraction: extract **1** bit starting at bit **b** from integer **n**

Table 34: Bitwise operators

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:

$$\mathrm{BIT}(n,\ b) = (n / 2^{b}) \backslash 2 \qquad \text{where} \qquad n \geq 0$$

$$\mathrm{BIT}(n,\ b) = 1 - \mathrm{BIT}((-n-1),\ b) \qquad \text{where} \qquad n < 0$$

Care must be taken whenever the infinitely long two's complement representation of a negative number is constructed. This representation will contain 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** ($\sim$) are defined on integers as follows, where b takes all values such that **b** ≥ **0**:

$$\mathrm{BIT}(i \wedge j,\ b) = \mathrm{BIT}(i,\ b) \times \mathrm{BIT}(j,\ b)$$

$$\mathrm{BIT}(i \vee j,\ b) = \mathrm{BIT}(i \wedge j,\ b) + \mathrm{BIT}(i \oplus j,\ b)$$

$$\mathrm{BIT}(i \oplus j,\ b) = (\mathrm{BIT}(i,\ b) + \mathrm{BIT}(j,\ b)) \backslash 2$$

$$\mathrm{BIT}(\sim i,\ b) = 1 - \mathrm{BIT}(i,\ b)$$

Note that bitwise **NOT** of any finite positive **i** will result 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 $\mathbf{b} \geq \mathbf{0}$ and $\mathbf{m} > \mathbf{0}$:

$$n_{\langle b \text{ FOR } m\rangle} = (n \gg b) \wedge (2^{m} - 1)$$

$$n_{\langle b\rangle} = n_{\langle b \text{ FOR } 1\rangle}$$

The result of $\mathbf{n}_{<b \text{ FOR } m>}$ is an integer in the range $[\mathbf{0}, \mathbf{2}^{\mathbf{m}})$.

16.3.4 Relational operators

Relational operators are defined to compare integral values and give a boolean result.

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**

Table 35: Relational operators

16.3.5 Boolean operators

Boolean operators are defined to 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.

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**

Table 36: Boolean operators

16.3.6 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 16.4.2: Assignment on page 118*.

Functions can contain side-effects.

Arithmetic functions

Function	Description
CountLeadingZeros(i)	Convert integer **i** to 32-bit bitfield and return the number of leading zeros in the bitfield. For example: If $\mathbf{i}_{<31>}$ is **1** then the return value is **0.** If all bits are **0** then the return value is **32**.

Table 37: Arithmetic functions

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.

Function	Description
$\text{ZeroExtend}_n(i)$	Convert integer **i** to an **n**-bit 2's complement unsigned range
$\text{SignExtend}_n(i)$	Convert integer **i** to an **n**-bit 2's complement signed range

Table 38: Integer conversion operators

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 39* shows the additional functions provided.

Operation	Description
Bit(i)	If **i** is a boolean, then 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 $\mathbf{i}_{\langle 0 \rangle}$
Register(i)	If **i** is a boolean, then 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 $\mathbf{i}_{\langle 0 \text{ FOR } 32 \rangle}$

Table 39: Conversion operators from integers to bit-fields

16.4 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 has a semi-colon terminator. 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.

16.4.1 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 any 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 will ensure that implemented behavior does not break the protection model. Thus, the implemented behavior will be some execution flow that is permitted for that user mode thread.

16.4.2 Assignment

The '←' operator is used 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 16.3: Expressions on page 112*. 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 one of these types.

Assignment to architectural state

This is where the variable is part of the architectural state (as described in *Table 40: Scalar architectural state on page 122*). 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

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 will hold a value which 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. Exploitation of implementation-defined values should be avoided to allow software to be portable between implementations.

Assignment of multiple values

Multi-value functions are used to return multiple values, and are only available when used in a multiple assignment context. The syntax 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 3 parameters can be represented as:

```
variable1, variable2 ←call(param1, param2, param3);
```

16.4.3 Conditional

Conditional behavior is specified using **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 then completes. If the condition is false, then any **ELSE IF** clauses are processed, in turn, in the same fashion. If no conditions are met and there is an **ELSE** clause then its block of statements 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 nearest **IF**.

For example:

```
IF (condition1)
  block1
ELSE IF (condition2)
  block2
ELSE
  block3
```

16.4.4 Repetition

Repetitive behavior is specified using 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 to within the iterated block. The **STEP s** can be omitted in which case the step-size takes the default value of **1**.

16.4.5 Exceptions

Exception handling is triggered by a **THROW** statement. 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 then none of the operations will update any architectural state.

If any operation in a bundle triggers an exception then an exception will be taken. The actions associated with the taking of an exception are described in *Section 5.2*.

There are two forms of throw statement:

```
THROW type;
```

and:

```
THROW type, value;
```

where **type** indicates the type of exception which is launched, and **value** is an optional argument to the exception handling sequence. If **value** is not given, then it is **UNDEFINED**.

The exception types and priorities are described in detail in *Chapter 5: Traps: exceptions and interrupts on page 31*.

16.4.6 Procedures

Procedure statements contain a procedure name followed by a list of comma-separated arguments contained within parentheses followed by a semi-colon. The execution of procedures typically causes side-effects to the architectural state of the machine.

Procedures are generally used where 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 will be given in the surrounding text.

An example procedure with two parameters is:

```
proc(param1, param2);
```

16.5 Architectural state

Chapter 3: Architectural state on page 25 contains a full description of the visible state. The notations used in the specification to refer to this state are summarized in *Table 40* and *Table 41*. 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.

Architectural state	Type is a bit-field containing:	Description
PC	32 bits	Program counter; address of the current bundle
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 R_0 reads as zero Assignments to R_0 are ignored
LR	32 bits	Link Register, synonym for R_{63}

Table 40: Scalar architectural state

Architectural state	Type is a bit-field containing:	Description
B_I where **i** is in **[0, 7]**	1 bit	8 x 1-bit Branch Registers

Table 40: Scalar 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 9: Control registers on page 65.*
MEM[i] where **i** is in **[0, 2^32)**	8 bits	2^{32} bytes of memory

Table 41: Array architectural state

16.6 Memory and control registers

16.6.1 Support functions

The following functions are used in the memory and control register descriptions.

Function	Description
DataBreakPoint(address)	Result is **TRUE** if **address** is in the range defined by data breakpoint control mechanism (*Chapter 6: Memory access protection units on page 39*), otherwise **FALSE**
Misaligned_n(address)	Result is **TRUE** if **address** is not **n**-bit aligned, otherwise **FALSE**
DPUNoTranslation(address)	Result is **TRUE** if the DPU has no mapping for **address**, otherwise **FALSE**
DPUSpecLoadRetZero(address)	Result is **TRUE** if the region containing **address** has the **S** bit of its attribute field set (*Chapter 6: Memory access protection units on page 39*), otherwise **FALSE**
ReadAccessViolation(address)	Result is **TRUE** if read access to **address** is not permitted by the DPU, otherwise **FALSE**
WriteAccessViolation(address)	Result is **TRUE** if write access to **address** is not permitted by the DPU, 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(address)	Result is **TRUE** if reading from **address** generates a Bus Error, otherwise **FALSE**
IsDBreakHit(address)	Result is **TRUE** if **address** will trigger a data breakpoint, otherwise it is **FALSE**

Table 42: Support functions

16.6.2 Memory model

The instruction specification uses a simple model of memory. It assumes, for example, that any caches are not architecturally visible. However, a fuller description of the behavior in other cases is defined by the text of the architecture manual.

Array slicing can be used to view an array as consisting of elements of a larger size. The notation **MEM[s FOR n]**, where **n** > **0**, denotes a memory slice containing the elements **MEM[s]**, **MEM[s+1]** through to **MEM[s+n-1]**. The type of this slice is a bit-field exactly large enough to contain a concatenation of the **n** selected elements. In this case it contain **8n** bits since the base type of **MEM** is byte.

The order of the concatenation depends on the endianness of the processor:

- If the processor is operating in a little endian mode, the concatenation order obeys the following condition as **i** (the byte number) varies in the range **[0, n)**:

  $$(\mathrm{MEM}[s \text{ FOR } n])_{\langle 8i \text{ FOR } 8\rangle} = \mathrm{MEM}[s+i]$$

  This equivalence states that byte number **i**, using little endian byte numbering (that is, byte **0** is bits **0** to **7**), in the bit-field **MEM[s FOR n]** is the **i**th. byte in memory counting upwards from **MEM[s]**.

- If the processor is operating in a big endian mode, the concatenation order obeys the following condition as **i** (the byte number) varies in the range **[0, n)**:

  $$(\mathrm{MEM}[s \text{ FOR } n])_{\langle 8(n-1-i) \text{ FOR } 8\rangle} = \mathrm{MEM}[s+i]$$

  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 **i**th. byte in memory counting upwards from **MEM[s]**.

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 42 on page 124*. These functions are used to model the behavior of the Data Protection Unit and Instruction Protection Unit described in *Chapter 6: Memory access protection units on page 39*.

Reading memory

The following functions are provided to support the reading of memory:

Function	Description
$ReadCheckMemory_n$(address)	Throws any non-BusError exception generated by an **n**-bit read from **address**
$ReadMemory_n$(address)	Issues an **n**-bit read to **address** (can generate BusError exception)
$DisReadCheckMemory_n$(address)	Throws any non-BusError exception generated by an **n**-bit dismissible read from **address**
$DisReadMemory_n$(address)	Returns either **n**-bits from **address** or **0** (can generate BusError exception)
ReadMemResponse()	Returns the value of the read request issued

Table 43: Memory read functions

The **$ReadCheckMemory_n$** procedure takes an integer parameter to indicate the address being accessed. The number of bits being read (**n**) is one of **8**, **16**, or **32**. The procedure throws any alignment or access violation exceptions generated by a read access to that address.

```
ReadCheckMemoryn(a);
```

is equivalent to:

```
IF (Misalignedn(a))
     THROW MISALIGNED_TRAP, a;
IF (PSW[DPU_ENABLE]) {
     IF (DPUNoTranslation(a))
          THROW DPU_NO_TRANSLATION, a;
     IF (ReadAccessViolation(a))
          THROW DPU_ACCESS_VIOLATION, a;
}
```

Similarly, if the memory access is a dismissible read:

```
DisReadCheckMemoryn(a);
```

is equivalent to:

```
IF (Misalignedn(a) AND PSW[SPECLOAD_MALIGNTRAP_EN])
     THROW MISALIGNED_TRAP, a;
IF (PSW[DPU_ENABLE] AND PSW[SPECLOAD_DPUTRAP_EN]) {
     IF (DPUNoTranslation(a))
          THROW DPU_NO_TRANSLATION, a;
     IF (ReadAccessViolation(a) AND NOT DPUSpecLoadRetZero(a))
          THROW DPU_ACCESS_VIOLATION, a;
}
```

The **ReadMemory**$_{\mathbf{n}}$ procedure takes an integer parameter to indicate the address being accessed. The number of bits being read (**n**) is one of **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 a sign-extension should be used when accessing the result using **ReadMemResponse**. The procedure call:

```
ReadMemoryn(a);
```

is equivalent to:

```
width ←n / 8;
IF (BusReadError(a))
     THROW BUS_DC_ERROR, a; // Non-recoverable
mem_response ←MEM[a FOR width];
```

The $\textbf{DisReadMemory}_n$ performs the same functionality for a dismissible read from memory. The procedure call:

```
DisReadMemoryn(a);
```

is equivalent to:

```
width ← n / 8;
IF (NOT DPUSpecLoadRetZero(a) AND
         NOT Misalignedn(a) AND
         NOT ReadAccessViolation(a)) {
     IF (BusReadError(a))
          THROW BUS_DC_ERROR, a; // Non-recoverable
     mem_response ←MEM[a FOR width];
}
ELSE
     mem_response ←0;
```

The function **ReadMemResponse** returns the data that will have 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.

Function	Description
PrefetchMemory(address)	Prefetch memory if possible.

Table 44: Memory prefetch procedure

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);
```

equivalent to:

```
IF (PSW[DPU_ENABLE])
    IF (NOT DPUNoTranslation(a) AND NOT ReadAccessViolation(a))
        Prefetch(a);
```

where **Prefetch** is a cache operation defined in *Section 16.6.4: Cache model on page 133*. Prefetching memory will not generate any exceptions.

Writing memory

The following procedures are provided to write memory.

Function	Description
WriteCheckMemory$_n$(address)	Throws any exception generated by an **n**-bit write to **address**
WriteMemory$_n$(address, value)	Aligned **n**-bit write to memory

Table 45: Memory write procedures

The **WriteCheckMemory$_n$** procedure takes an integer parameter to indicate the address being accessed. The number of bits being written (**n**) is one of **8**, **16**, or **32**. The procedure throws any alignment or access violation exceptions generated by a write access to that address.

```
WriteCheckMemoryn(a);
```

is equivalent to:

```
IF (Misalignedn(a))
     THROW MISALIGNED_TRAP, a;
IF (PSW[DPU_ENABLE]) {
     IF (DPUNoTranslation(a))
          THROW DPU_NO_TRANSLATION, a;
     IF (WriteAccessViolation(a))
          THROW DPU_ACCESS_VIOLATION, a;
}
```

The **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 one of **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:

```
WriteMemoryn(a, value);
```

is equivalent to:

```
width ←n / 8;
MEM[a FOR width] ←value<0 FOR n>;
```

16.6.3 Control register model

Reading control registers

The following procedures are provided to read from control registers.

Note: Only word (32-bit) control register accesses are supported.

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**

Table 46: Control register read functions

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 to read from control registers. Note that only word (32-bit) control register accesses are supported

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**

Table 47: Control registers write procedures

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);
```

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;
```

16.6.4 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 7: Memory subsystem on page 47*.

Procedure	Description
PurgeIns( )	Invalidate the entire instruction cache
Sync( )	Data memory subsystem synchronization function
PurgeAddress(address)	Purge **address** from the data cache
PurgeSet(address)	Purge a set of lines from the data cache
Prefetch(address)	Prefetch a data cache line if it is in cachable memory

Table 48: Procedures to model cache operations

Instruction set

17

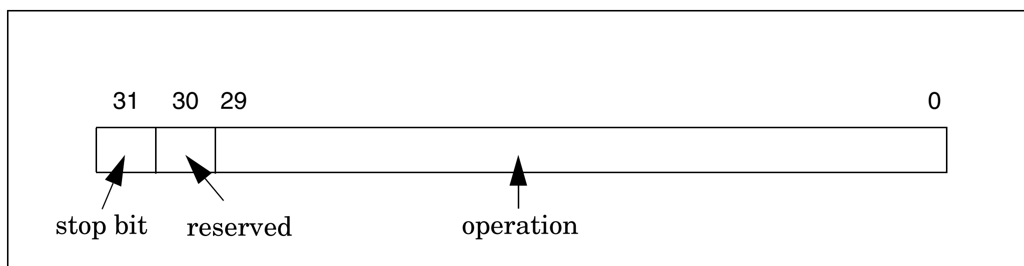
17.1 Introduction

This chapter contains descriptions of all the operations and macros (pseudo-operations) in the ST220 instruction set. *Section 17.2* has been included in order to describe how operations are encoded in the context of bundles.

17.2 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. The most significant bit of each syllable (bit 31) is a **stop bit** which is set to indicate that it is the last in the bundle.

A syllable will therefore look like:



17.2.1 Extended immediates

Many operations have an **Immediate** form. In general only small (9-bit) immediates can be directly encoded in a single word syllable. In the event that larger immediates are required, an immediate extension is used. This extension is encoded in an adjacent word in the bundle, making the operation effectively a two-word operation.

These immediate extensions associate either with the operation to their left or their right in the bundle. Bit 23 is used to indicate the association:

	31	30	29 … 24	23	22 … 0
imml	s		010101	0	extension
immr	s		010101	1	extension

Figure 15:

The semantic descriptions of **Immediate** form operations use the following function to take into account possible immediate extensions:

Function	Description
Imm(i)	Given short immediate value **i**, returns an integer value that represents the full immediate.

Table 49: Extended immediate functions

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:

(ZeroExtend$_{23}$(extension) << 9) + ZeroExtend$_{9}$(i);

Where **extension** represents the lower 23 bits of the associated **immr** or **imml**.

Otherwise **Imm** returns:

SignExtend$_{9}$(i);

17.2.2 Encoding restrictions

There are a number of restrictions placed on the encoding of bundles. It is the duty of the assembler to ensure that these restriction are obeyed.

1. Long immediates must be encoded at even word addresses.
2. Multiply operations must be encoded at odd word addresses.
3. There may only be one control flow operation per bundle, and it must be the first syllable.
4. There may only be one load or store operation per bundle.

17.3 Operation specifications

The specification of each operation contains the following fields:

- Name: The name of the operation with an optional subscript. The subscript is used to distinguish between operations with different operand types. For example, there are **Register** and **Immediate** format integer operations. If no subscript is used, then there is only one format for the operation.
- Syntax: Presents the assembly syntax of the operation (*ST200 Programming Manual*).
- 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 will either be register designators or immediate constants.
- Semantics: A table containing the statements (*Section 16.4*) that define the operation. The notation used is defined in *Chapter 16 on page 109*. The table is divided into two parts by the commit point:

Phase	
Pre-commit phase: • No architectural state of the machine is updated. • Any recoverable exceptions will be thrown here.	←Commit point
Commit phase - executed if no exceptions have been thrown: • All architectural state is updated. • Any exceptions thrown here are non-recoverable[a].	

a. For the ST220 the only non-recoverable exception is a bus error.

- Description: A brief textual description of the operation.
- Restrictions: Contains any details of restrictions, these 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 17.2.2*.
    - 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 will detail how and when they are thrown.

17.4 Example operations

17.4.1 add Immediate

The specification for this operation is shown below:

add Immediate

add R_{IDEST} = R_{SRC1}, *ISRC2*

s		0010	00000	*ISRC2*	*IDEST*	*SRC1*
31	30	29–26	25–21	20–12	11–6	5–0

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(*ISRC2*));
result ←operand1 + operand2;

R_{IDEST} ←Register(result);

Description:

Add

Restrictions:

No address/bundle restrictions.
No latency constraints.

The operation is given the subscript **Immediate** to indicate that one of its source operands is an immediate rather than both 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:

- The opcode: Bits 29:21.
- The operands: An **s** in bit 31 represents the stop bit (*Section 17.2*).
  - The 9-bit immediate constant, bits 20:12.
  - The destination register designator, bits 11:6.
  - The source register designator, bits 5:0.
- Unused bits: Bit 30.

The semantics table specifies the effects of the executing this operation. The table is divided into two parts. The first half containing statements which do not affect the architectural state of the machine. The second half containing statements that will not be executed if an exception occurs in the bundle.

The statements themselves are organized into 3 stages as follows:

1 The first 2 statements read the required source information:

   operand1 $\leftarrow$**SignExtend**$_{32}$**(**R_{SRC1}**);**
   operand2 $\leftarrow$**SignExtend**$_{32}$**(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 17.2.1*). 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 $\leftarrow$**operand1 + operand2;**

   This statement does not refer to any architectural state. It adds the 2 integers **operand1** and **operand2** together, and assigns the result to a temporary integer called **result**. Note that 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:

R_{IDEST} ←**Register(result);**

The function **Register** (*Section 16.3.6*) converts the integer **result** back to a bit-field, discarding any redundant higher bits. This value is then assigned to the $\mathbf{R}_{IDEST}$ register.

After the semantic description is a simple textual description of the operation.

The section listing restrictions for this operation shows that it has no restrictions at all. This means that up to four of these operations can be used in a bundle, and that all operands will be ready for use by operations in the next bundle.

Finally, this operation can not generate any exceptions.

17.5 Macros

The following are the currently implemented pseudo-operations.

<table>
<tr><th></th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>nop</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">00000</td><td colspan="3"></td><td colspan="6">000000</td><td colspan="6">000000</td><td colspan="6">000000</td></tr>
<tr><td>mov</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">00000</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">000000</td></tr>
<tr><td>mtb</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>1</td><td>1</td><td colspan="4">1100</td><td colspan="3">BDEST</td><td colspan="6"></td><td colspan="6">000000</td><td colspan="6">SRC1</td></tr>
<tr><td>mov</td><td>s</td><td></td><td colspan="2">00</td><td>1</td><td>0</td><td colspan="5">00000</td><td colspan="9">ISRC2</td><td colspan="6">IDEST</td><td colspan="6">000000</td></tr>
<tr><td>zxtb</td><td>s</td><td></td><td colspan="2">00</td><td>1</td><td>0</td><td colspan="5">01001</td><td colspan="9">011111111</td><td colspan="6">IDEST</td><td colspan="6">SRC1</td></tr>
<tr><td>mfb</td><td>s</td><td></td><td colspan="2">01</td><td>1</td><td colspan="3">001</td><td colspan="3">SCOND</td><td colspan="9">000000001</td><td colspan="6">IDEST</td><td colspan="6">000000</td></tr>
<tr><td>syncins</td><td>s</td><td></td><td colspan="2">11</td><td>0</td><td colspan="3">001</td><td>0</td><td colspan="23">00000000000000000000001</td></tr>
</table>

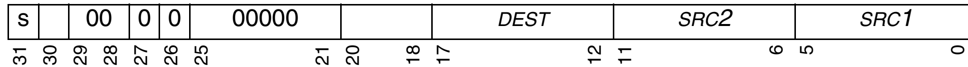
Figure 16: Macros

17.6 Operations

Each operation is now specified. They are listed alphabetically for ease of use. The semantics of the operations are written using the notational language defined in *Chapter 16 on page 109*.

add Register

add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 + operand2;
R_{DEST} ←Register(result);

Description:

Add

Restrictions:

No address/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*)); result ←operand1 + operand2;
R_{IDEST} ←Register(result);

Description:

Add

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

addcg

addcg R_{DEST}, B_{BDEST} = R_{SRC1}, R_{SRC2}, B_{SCOND}

s		01	0010	*SCOND*	*BDEST*	*DEST*	*SRC2*	*SRC1*
31	30	29 28	27 24	23 21	20 18	17 12	11 6	5 0

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); carryin ←ZeroExtend$_1$(B_{SCOND}); result ←(operand1 + operand2) + carryin; carryout ←Bit(result, 32);
R_{DEST} ←Register(result); B_{BDEST} ←Bit(carryout);

Description:

Add with carry and generate carry

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

and Register

and R_{DEST} = R_{SRC1}, R_{SRC2}

s		00	0	0	01001		DEST	SRC2	SRC1
31	30	29 28	27	26	25 21	20 18	17 12	11 6	5 0

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 ∧ operand2;
R_{DEST} ←Register(result);

Description:

Bitwise and

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

and Immediate

and R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ← $SignExtend_{32}(R_{SRC1})$; operand2 ← $SignExtend_{32}(Imm(ISRC2))$; result ← operand1 ∧ operand2;
R_{IDEST} ← Register(result);

Description:

Bitwise and

Restrictions:

No address/bundle restrictions.

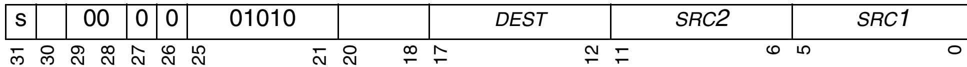
No latency constraints.

Exceptions:

None.

andc Register

andc $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←(~ operand1) ∧ operand2;
R_{DEST} ←Register(result);

Description:

Complement and bitwise and

Restrictions:

No address/bundle restrictions.

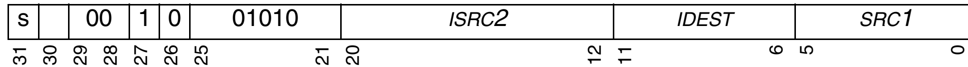
No latency constraints.

Exceptions:

None.

andc Immediate

andc R_{IDEST} = R_{SRC1}, *ISRC2*



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←(~ operand1) ∧ operand2;
R_{IDEST} ←Register(result);

Description:

Complement and bitwise and

Restrictions:

No address/bundle restrictions.

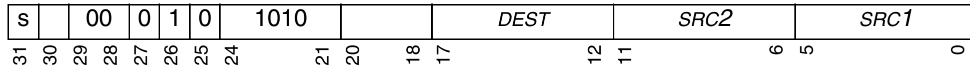
No latency constraints.

Exceptions:

None.

andl Register - Register

andl R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←(operand1 ≠ 0) AND (operand2 ≠ 0);
R_{DEST} ←Register(result);

Description:

Logical and

Restrictions:

No address/bundle restrictions.

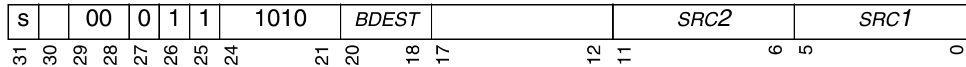
No latency constraints.

Exceptions:

None.

andl Branch Register - Register

andl B_{BDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←(operand1 ≠ 0) AND (operand2 ≠ 0);
B_{BDEST} ←Bit(result);

Description:

Logical and

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

andl Register - Immediate

andl R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←(operand1 ≠ 0) AND (operand2 ≠ 0);
R_{IDEST} ←Register(result);

Description:

Logical and

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

andl Branch Register - Immediate

andl B_{IBDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←(operand1 ≠ 0) AND (operand2 ≠ 0);
B_{IBDEST} ←Bit(result);

Description:

Logical and

Restrictions:

No address/bundle restrictions.

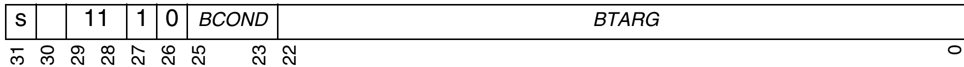
No latency constraints.

Exceptions:

None.

br

br B_{BCOND}, *BTARG*



Semantics:

```
operand1 ←ZeroExtend1(B_BCOND);
operand2 ←SignExtend23(BTARG);
IF (operand1 ≠ 0)
        PC ←Register(ZeroExtend32(BUNDLE_PC) + (operand2 << 2));
```

Description:

Branch

Restrictions:

Must be the first syllable of a bundle.

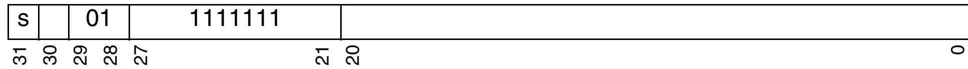
Instructions writing B_{BCOND} must be followed by 2 bundles before this instruction can be issued.

Exceptions:

None.

break

break



Semantics:

THROW ILL_INST;

Description:

Break

Restrictions:

No address/bundle restrictions.

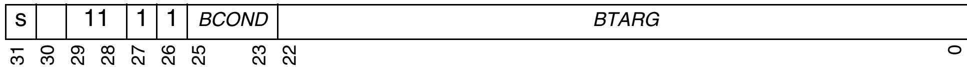
No latency constraints.

Exceptions:

ILL_INST

brf

brf B_{BCOND}, *BTARG*



Semantics:

```
operand1 ←ZeroExtend1(B_BCOND);
operand2 ←SignExtend23(BTARG);
IF (operand1 = 0)
        PC ←Register(ZeroExtend32(BUNDLE_PC) + (operand2 << 2));
```

Description:

Branch false

Restrictions:

Must be the first syllable of a bundle.

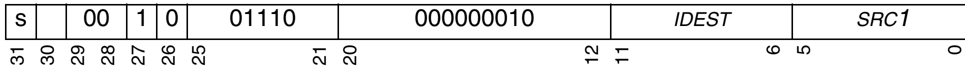
Instructions writing B_{BCOND} must be followed by 2 bundles before this instruction can be issued.

Exceptions:

None.

bswap

bswap R_{IDEST} = R_{SRC1}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); byte0 ←operand1$_{<0 \text{ FOR } 8>}$; byte1 ←operand1$_{<8 \text{ FOR } 8>}$; byte2 ←operand1$_{<16 \text{ FOR } 8>}$; byte3 ←operand1$_{<24 \text{ FOR } 8>}$; result ←((byte0 << 24) ∨ (byte1 << 16)) ∨ ((byte2 << 8) ∨ byte3);
R_{IDEST} ←Register(result);

Description:

Byte Swap

Restrictions:

No address/bundle restrictions.

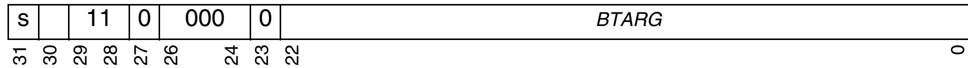
No latency constraints.

Exceptions:

None.

call Immediate

call LR = *BTARG*



Semantics:

```
operand1 ←SignExtend23(BTARG);
NEXT_PC ←PC;
PC ←Register(ZeroExtend32(BUNDLE_PC) + (operand1 << 2));
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 LR = LR

s		11	0	000	1	
31	30	29 28	27	26 24	23	22 0

Semantics:

```
NEXT_PC ←PC;
PC ←Register(ZeroExtend32(LR));
LR ←NEXT_PC;
```

Description:

Jump (using Link Register) and link

Restrictions:

Must be the first syllable of a bundle.

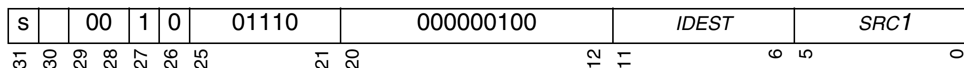
Instructions writing LR must be followed by 3 bundles before this instruction can be issued.

Exceptions:

None.

clz

clz R_{IDEST} = R_{SRC1}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); result ←CountLeadingZeros(operand1);
R_{IDEST} ←Register(result);

Description:

Count leading zeros

Restrictions:

No address/bundle restrictions.

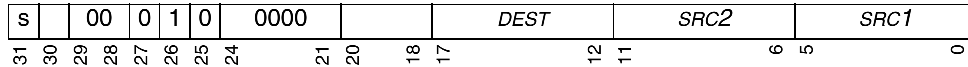
No latency constraints.

Exceptions:

None.

cmpeq Register - Register

cmpeq R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 = operand2;
R_{DEST} ←Register(result);

Description:

Test for equality

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

cmpeq Branch Register - Register

cmpeq B_{BDEST} = R_{SRC1}, R_{SRC2}

s		00	0	1	1	0000	*BDEST*		*SRC2*	*SRC1*
31	30	29 28	27	26	25	24 21	20 18	17 12	11 6	5 0

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 = operand2;
B_{BDEST} ←Bit(result);

Description:

Test for equality

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

cmpeq Register - Immediate

cmpeq R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←operand1 = operand2;
R_{IDEST} ←Register(result);

Description:

Test for equality

Restrictions:

No address/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*)); result ←operand1 = operand2;
B_{IBDEST} ←Bit(result);

Description:

Test for equality

Restrictions:

No address/bundle restrictions.

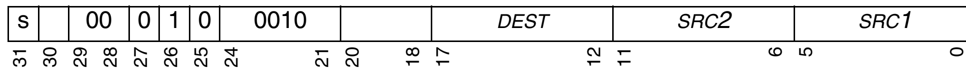
No latency constraints.

Exceptions:

None.

cmpge Register - Register

cmpge R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 ≥ operand2;
R_{DEST} ←Register(result);

Description:

Signed compare equal or greater than

Restrictions:

No address/bundle restrictions.

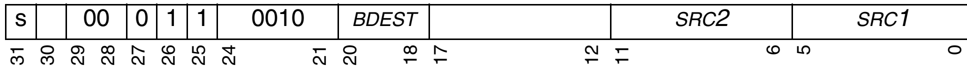
No latency constraints.

Exceptions:

None.

cmpge Branch Register - Register

cmpge B_{BDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 ≥ operand2;
B_{BDEST} ←Bit(result);

Description:

Signed compare equal or greater than

Restrictions:

No address/bundle restrictions.

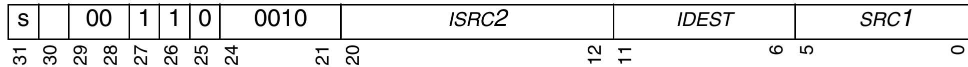
No latency constraints.

Exceptions:

None.

cmpge Register - Immediate

cmpge R_{IDEST} = R_{SRC1}, *ISRC2*



31	30	29–28	27	26	25	24–21	20–12	11–6	5–0
s		00	1	1	0	0010	*ISRC2*	*IDEST*	*SRC1*

Semantics:

operand1 ←$\text{SignExtend}_{32}(R_{SRC1})$; operand2 ←$\text{SignExtend}_{32}(\text{Imm}(ISRC2))$; result ←operand1 ≥ operand2;
R_{IDEST} ←Register(result);

Description:

Signed compare equal or greater than

Restrictions:

No address/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*)); result ←operand1 ≥ operand2;
B_{IBDEST} ←Bit(result);

Description:

Signed compare equal or greater than

Restrictions:

No address/bundle restrictions.

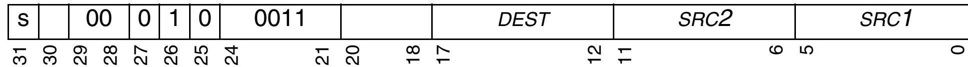
No latency constraints.

Exceptions:

None.

cmpgeu Register - Register

cmpgeu R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result ←operand1 ≥ operand2;
R_{DEST} ←Register(result);

Description:

Unsigned compare equal or greater than

Restrictions:

No address/bundle restrictions.

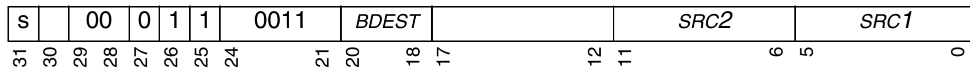
No latency constraints.

Exceptions:

None.

cmpgeu Branch Register - Register

cmpgeu B_{BDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result ←operand1 ≥ operand2;
B_{BDEST} ←Bit(result);

Description:

Unsigned compare equal or greater than

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

cmpgeu Register - Immediate

cmpgeu R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(Imm(*ISRC2*)); result ←operand1 ≥ operand2;
R_{IDEST} ←Register(result);

Description:

Unsigned compare equal or greater than

Restrictions:

No address/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*)); result ←operand1 ≥ operand2;
B_{IBDEST} ←Bit(result);

Description:

Unsigned compare equal or greater than

Restrictions:

No address/bundle restrictions.

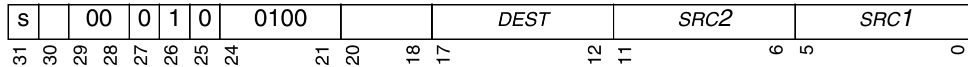
No latency constraints.

Exceptions:

None.

cmpgt Register - Register

cmpgt R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 > operand2;
R_{DEST} ←Register(result);

Description:

Signed compare greater than

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

cmpgt Branch Register - Register

cmpgt B_{BDEST} = R_{SRC1}, R_{SRC2}

s		00	0	1	1	0100	BDEST		SRC2	SRC1
31	30	29 28	27	26	25	24 21	20 18	17 12	11 6	5 0

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 > operand2;
B_{BDEST} ←Bit(result);

Description:

Signed compare greater than

Restrictions:

No address/bundle restrictions.

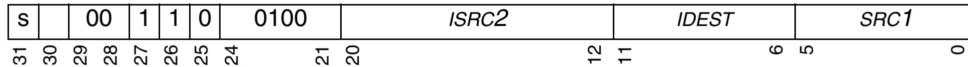
No latency constraints.

Exceptions:

None.

cmpgt Register - Immediate

cmpgt R_{IDEST} = R_{SRC1}, *ISRC2*



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←operand1 > operand2;
R_{IDEST} ←Register(result);

Description:

Signed compare greater than

Restrictions:

No address/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*)); result ←operand1 > operand2;
B_{IBDEST} ←Bit(result);

Description:

Signed compare greater than

Restrictions:

No address/bundle restrictions.

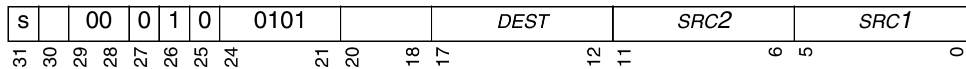
No latency constraints.

Exceptions:

None.

cmpgtu Register - Register

cmpgtu R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result ←operand1 > operand2;
R_{DEST} ←Register(result);

Description:

Unsigned compare greater than

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

cmpgtu Branch Register - Register

cmpgtu $B_{BDEST} = R_{SRC1}, R_{SRC2}$

s		00	0	1	1	0101	BDEST		SRC2	SRC1
31	30	29 28	27	26	25	24 21	20 18	17 12	11 6	5 0

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result ←operand1 > operand2;
B_{BDEST} ←Bit(result);

Description:

Unsigned compare greater than

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

cmpgtu Register - Immediate

cmpgtu R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(Imm(*ISRC2*)); result ←operand1 > operand2;
R_{IDEST} ←Register(result);

Description:

Unsigned compare greater than

Restrictions:

No address/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*)); result ←operand1 > operand2;
B_{IBDEST} ←Bit(result);

Description:

Unsigned compare greater than

Restrictions:

No address/bundle restrictions.

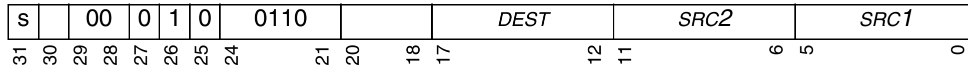
No latency constraints.

Exceptions:

None.

cmple Register - Register

cmple R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 ≤operand2;
R_{DEST} ←Register(result);

Description:

Signed compare equal or less than

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

cmple Branch Register - Register

cmple B_{BDEST} = R_{SRC1}, R_{SRC2}

s		00	0	1	1	0110	*BDEST*		*SRC2*	*SRC1*
31	30	29 28	27	26	25	24 21	20 18	17 12	11 6	5 0

Semantics:

operand1 ← SignExtend$_{32}$(R_{SRC1}); operand2 ← SignExtend$_{32}$(R_{SRC2}); result ← operand1 ≤ operand2;
B_{BDEST} ← Bit(result);

Description:

Signed compare equal or less than

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

cmple Register - Immediate

cmple R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←operand1 ≤operand2;
R_{IDEST} ←Register(result);

Description:

Signed compare equal or less than

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

cmple Branch Register - Immediate

cmple $B_{IBDEST} = R_{SRC1}$, *ISRC2*

Semantics:

operand1 ← $\text{SignExtend}_{32}(R_{SRC1})$; operand2 ← $\text{SignExtend}_{32}(\text{Imm}(ISRC2))$; result ← operand1 ≤ operand2;
B_{IBDEST} ← Bit(result);

Description:

Signed compare equal or less than

Restrictions:

No address/bundle restrictions.

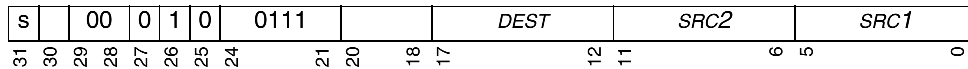
No latency constraints.

Exceptions:

None.

cmpleu Register - Register

cmpleu R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result ←operand1 ≤operand2;
R_{DEST} ←Register(result);

Description:

Unsigned compare equal or less than

Restrictions:

No address/bundle restrictions.

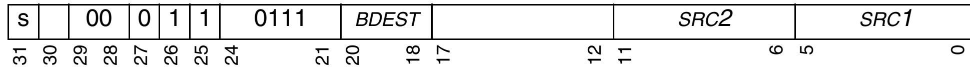
No latency constraints.

Exceptions:

None.

cmpleu Branch Register - Register

cmpleu B_{BDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result ←operand1 ≤operand2;
B_{BDEST} ←Bit(result);

Description:

Unsigned compare equal or less than

Restrictions:

No address/bundle restrictions.

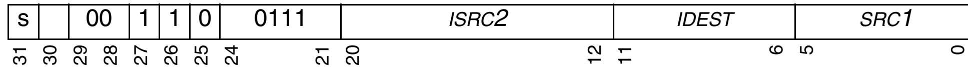
No latency constraints.

Exceptions:

None.

cmpleu Register - Immediate

cmpleu R_{IDEST} = R_{SRC1}, *ISRC2*



Semantics:

operand1 ← ZeroExtend$_{32}$(R_{SRC1}); operand2 ← ZeroExtend$_{32}$(Imm(*ISRC2*)); result ← operand1 ≤ operand2;
R_{IDEST} ← Register(result);

Description:

Unsigned compare equal or less than

Restrictions:

No address/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*)); result ←operand1 ≤operand2;
B_{IBDEST} ←Bit(result);

Description:

Unsigned compare equal or less than

Restrictions:

No address/bundle restrictions.

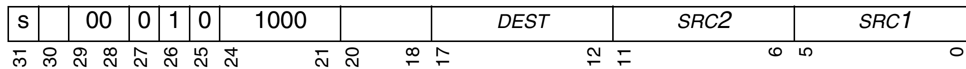
No latency constraints.

Exceptions:

None.

cmplt Register - Register

cmplt R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 < operand2;
R_{DEST} ←Register(result);

Description:

Signed compare less than

Restrictions:

No address/bundle restrictions.

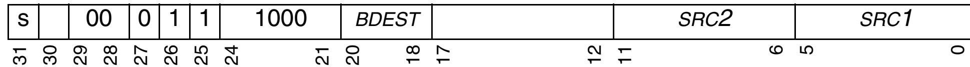
No latency constraints.

Exceptions:

None.

cmplt Branch Register - Register

cmplt B_{BDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 < operand2;
B_{BDEST} ←Bit(result);

Description:

Signed compare less than

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

cmplt Register - Immediate

cmplt R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(*ISRC2*));
result ←operand1 < operand2;

R_{IDEST} ←Register(result);

Description:

Signed compare less than

Restrictions:

No address/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*)); result ←operand1 < operand2;
B_{IBDEST} ←Bit(result);

Description:

Signed compare less than

Restrictions:

No address/bundle restrictions.

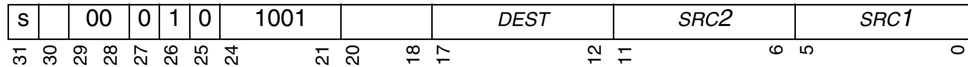
No latency constraints.

Exceptions:

None.

cmpltu Register - Register

cmpltu $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result ←operand1 < operand2;
R_{DEST} ←Register(result);

Description:

Unsigned compare less than

Restrictions:

No address/bundle restrictions.

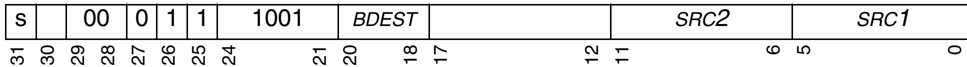
No latency constraints.

Exceptions:

None.

cmpltu Branch Register - Register

cmpltu B_{BDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(R_{SRC2}); result ←operand1 < operand2;
B_{BDEST} ←Bit(result);

Description:

Unsigned compare less than

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

cmpltu Register - Immediate

cmpltu R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{32}$(Imm(*ISRC2*)); result ←operand1 < operand2;
R_{IDEST} ←Register(result);

Description:

Unsigned compare less than

Restrictions:

No address/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*)); result ←operand1 < operand2;
B_{IBDEST} ←Bit(result);

Description:

Unsigned compare less than

Restrictions:

No address/bundle restrictions.

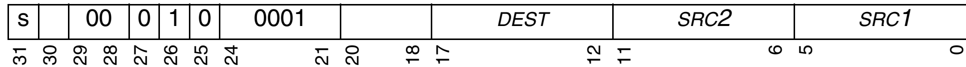
No latency constraints.

Exceptions:

None.

cmpne Register - Register

cmpne R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 ≠ operand2;
R_{DEST} ←Register(result);

Description:

Test for inequality

Restrictions:

No address/bundle restrictions.

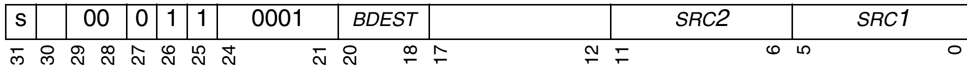
No latency constraints.

Exceptions:

None.

cmpne Branch Register - Register

cmpne B_{BDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 ≠ operand2;
B_{BDEST} ←Bit(result);

Description:

Test for inequality

Restrictions:

No address/bundle restrictions.

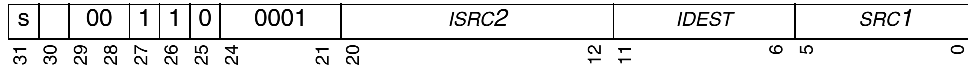
No latency constraints.

Exceptions:

None.

cmpne Register - Immediate

cmpne R_{IDEST} = R_{SRC1}, *ISRC2*



s		00	1	1	0	0001	*ISRC2*	*IDEST*	*SRC1*
31	30	29 28	27	26	25	24 21	20 12	11 6	5 0

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(*ISRC2*));
result ←operand1 ≠ operand2;

R_{IDEST} ←Register(result);

Description:

Test for inequality

Restrictions:

No address/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*)); result ←operand1 ≠ operand2;
B_{IBDEST} ←Bit(result);

Description:

Test for inequality

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

divs

divs R_{DEST}, B_{BDEST} = R_{SRC1}, R_{SRC2}, B_{SCOND}

s		01	0100	*SCOND*	*BDEST*	*DEST*	*SRC2*	*SRC1*
31	30	29 28	27 24	23 21	20 18	17 12	11 6	5 0

Semantics:

```
operand1 ←SignExtend32(R_SRC1);
operand2 ←SignExtend32(R_SRC2);
operand3 ←ZeroExtend1(B_SCOND);
temp ←ZeroExtend32(operand1 × 2) ∨  (operand3 ∧ 1);
IF (operand1 < 0)
 {
          result ←temp + operand2;
          quotientBit ←1;
 }
ELSE
 {
          result ←temp - operand2;
          quotientBit ←0;
 }
```

```
R_DEST ←Register(result);
B_BDEST ←Bit(quotientBit);
```

Description:

Non-restoring divide stage

Restrictions:

No address/bundle restrictions.

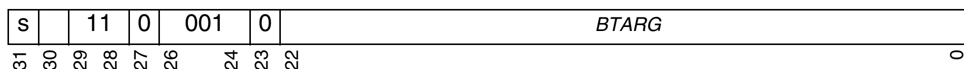
No latency constraints.

Exceptions:

None.

goto Immediate

goto *BTARG*



Semantics:

operand1 ←SignExtend$_{23}$(*BTARG*);
PC ←Register(ZeroExtend$_{32}$(BUNDLE_PC) + (operand1 << 2));

Description:

Jump

Restrictions:

Must be the first syllable of a bundle.

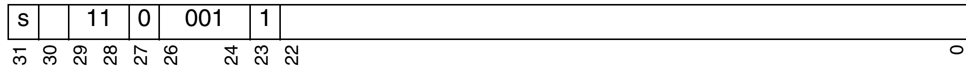
No latency constraints.

Exceptions:

None.

goto Link Register

goto LR



Semantics:

PC ←Register(ZeroExtend$_{32}$(LR));

Description:

Jump (using Link Register)

Restrictions:

Must be the first syllable of a bundle.

Instructions writing LR must be followed by 3 bundles before this instruction can be issued.

Exceptions:

None.

imml

imml imm

31	30	29–28	27–23	22–0
s		01	01010	*IMM*

Semantics:

extension ←ZeroExtend_{23}(imm);

Description:

Long immediate for previous syllable

Restrictions:

Must be encoded at even word addresses.

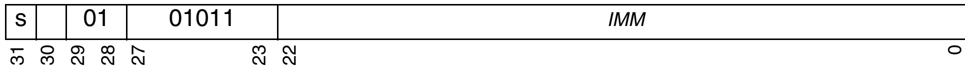
No latency constraints.

Exceptions:

None.

immr

immr imm



Semantics:

extension ←ZeroExtend$_{23}$(imm);

Description:

Long immediate for next syllable

Restrictions:

Must be encoded at even word addresses.

No latency constraints.

Exceptions:

None.

ldb

ldb R_{IDEST} = *ISRC2*[R_{SRC1}]

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOLATION;
ReadCheckMemory8(ea);
```

```
ReadMemory8(ea);
result ←SignExtend8(ReadMemResponse());
RIDEST ←Register(result);
```

Description:

Signed load byte

Restrictions:

Cannot write the link register (LR).

Uses the load/store unit, for which only one operation is allowed per bundle.

This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

DBREAK

CREG_ACCESS_VIOLATION

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldb.d

ldb.d R_{IDEST} = *ISRC2*[R_{SRC1}]

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (NOT IsCRegSpace(ea))
        DisReadCheckMemory8(ea);
```

```
IF (NOT IsCRegSpace(ea))
        DisReadMemory8(ea);
IF (IsCRegSpace(ea))
        result ←0;
ELSE
        result ←SignExtend8(ReadMemResponse());
RIDEST ←Register(result);
```

Description:

Signed load byte dismissable

Restrictions:

Cannot write the link register (LR).

Uses the load/store unit, for which only one operation is allowed per bundle.

This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

DBREAK

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldbu

ldbu R_{IDEST} = *ISRC2*[R_{SRC1}]

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOLATION;
ReadCheckMemory8(ea);
```

```
ReadMemory8(ea);
result ←ZeroExtend8(ReadMemResponse());
RIDEST ←Register(result);
```

Description:

Unsigned load byte

Restrictions:

Cannot write the link register (LR).

Uses the load/store unit, for which only one operation is allowed per bundle.

This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

DBREAK

CREG_ACCESS_VIOLATION

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldbu.d

ldbu.d R_{IDEST} = *ISRC2*[R_{SRC1}]

31	30	29–28	27–24	23	22	21	20–12	11–6	5–0
s		10	0100	1			*ISRC2*	*IDEST*	*SRC1*

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (NOT IsCRegSpace(ea))
        DisReadCheckMemory8(ea);
```

```
IF (NOT IsCRegSpace(ea))
        DisReadMemory8(ea);
IF (IsCRegSpace(ea))
        result ←0;
ELSE
        result ←ZeroExtend8(ReadMemResponse());
RIDEST ←Register(result);
```

Description:

Unsigned load byte dismissable

Restrictions:

Cannot write the link register (LR).

Uses the load/store unit, for which only one operation is allowed per bundle.

This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

DBREAK

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldh

ldh R_{IDEST} = *ISRC2*[R_{SRC1}]

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOLATION;
ReadCheckMemory16(ea);
```

```
ReadMemory16(ea);
result ←SignExtend16(ReadMemResponse());
RIDEST ←Register(result);
```

Description:

Signed load half-word

Restrictions:

Cannot write the link register (LR).

Uses the load/store unit, for which only one operation is allowed per bundle.

This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

DBREAK

CREG_ACCESS_VIOLATION

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldh.d

ldh.d R_{IDEST} = *ISRC2*[R_{SRC1}]

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (NOT IsCRegSpace(ea))
        DisReadCheckMemory16(ea);
```

```
IF (NOT IsCRegSpace(ea))
        DisReadMemory16(ea);
IF (IsCRegSpace(ea))
        result ←0;
ELSE
        result ←SignExtend16(ReadMemResponse());
RIDEST ←Register(result);
```

Description:

Signed load half-word dismissable

Restrictions:

Cannot write the link register (LR).

Uses the load/store unit, for which only one operation is allowed per bundle.

This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

DBREAK

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldhu

ldhu R_{IDEST} = *ISRC2*[R_{SRC1}]

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOLATION;
ReadCheckMemory16(ea);
```

```
ReadMemory16(ea);
result ←ZeroExtend16(ReadMemResponse());
RIDEST ←Register(result);
```

Description:

Unsigned load half-word

Restrictions:

Cannot write the link register (LR).

Uses the load/store unit, for which only one operation is allowed per bundle.

This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

DBREAK

CREG_ACCESS_VIOLATION

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldhu.d

ldhu.d R_{IDEST} = *ISRC2*[R_{SRC1}]

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (NOT IsCRegSpace(ea))
        DisReadCheckMemory16(ea);
```

```
IF (NOT IsCRegSpace(ea))
        DisReadMemory16(ea);
IF (IsCRegSpace(ea))
        result ←0;
ELSE
        result ←ZeroExtend16(ReadMemResponse());
RIDEST ←Register(result);
```

Description:

Unsigned load half-word dismissable

Restrictions:

Cannot write the link register (LR).

Uses the load/store unit, for which only one operation is allowed per bundle.

This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

DBREAK

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldw

ldw R_{IDEST} = *ISRC2*[R_{SRC1}]

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        ReadCheckCReg(ea);
ReadCheckMemory32(ea);
```

```
IF (IsCRegSpace(ea))
        ReadCReg(ea);
ELSE
        ReadMemory32(ea);
result ←SignExtend32(ReadMemResponse());
RIDEST ←Register(result);
```

Description:

Load word

Restrictions:

Uses the load/store unit, for which only one operation is allowed per bundle.

This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

CREG_NO_MAPPING

CREG_ACCESS_VIOLATION

ldw.d

ldw.d R_{IDEST} = *ISRC2*[R_{SRC1}]

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (NOT IsCRegSpace(ea))
        DisReadCheckMemory32(ea);
```

```
IF (NOT IsCRegSpace(ea))
        DisReadMemory32(ea);
IF (IsCRegSpace(ea))
        result ←0;
ELSE
        result ←SignExtend32(ReadMemResponse());
RIDEST ←Register(result);
```

Description:

Load word dismissable

Restrictions:

Uses the load/store unit, for which only one operation is allowed per bundle.

This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

DBREAK

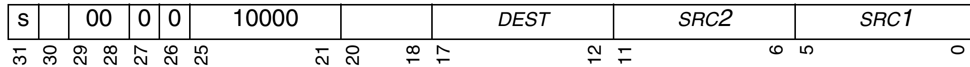
MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

max Register

max R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
IF (operand1 > operand2)
        result ←operand1;
ELSE
        result ←operand2;
```

```
RDEST ←Register(result);
```

Description:

Signed maximum

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

max Immediate

max R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(Imm(*ISRC2*));
IF (operand1 > operand2)
    result ←operand1;
ELSE
    result ←operand2;

R_{IDEST} ←Register(result);

Description:

Signed maximum

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

maxu Register

maxu $R_{DEST} = R_{SRC1}, R_{SRC2}$

s		00	0	0	10001		DEST	SRC2	SRC1
31	30	29 28	27	26	25 21	20 18	17 12	11 6	5 0

Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
IF (operand1 > operand2)
        result ←operand1;
ELSE
        result ←operand2;
```

```
RDEST ←Register(result);
```

Description:

Unsigned maximum

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

maxu Immediate

maxu R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

```
operand1 ←ZeroExtend32(R_SRC1);
operand2 ←ZeroExtend32(Imm(ISRC2));
IF (operand1 > operand2)
        result ←operand1;
ELSE
        result ←operand2;
```

```
R_IDEST ←Register(result);
```

Description:

Unsigned maximum

Restrictions:

No address/bundle restrictions.

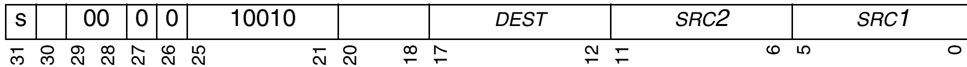
No latency constraints.

Exceptions:

None.

min Register

min R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

```
operand1 ←SignExtend32(RSRC1);
operand2 ←SignExtend32(RSRC2);
IF (operand1 < operand2)
        result ←operand1;
ELSE
        result ←operand2;
```

```
RDEST ←Register(result);
```

Description:

Signed minimum

Restrictions:

No address/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)
        result ←operand1;
ELSE
        result ←operand2;
```

```
RIDEST ←Register(result);
```

Description:

Signed minimum

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

minu Register

minu R_{DEST} = R_{SRC1}, R_{SRC2}

s		00	0	0	10011		DEST	SRC2	SRC1
31	30	29 28	27	26	25 21	20 18	17 12	11 6	5 0

Semantics:

```
operand1 ←ZeroExtend32(RSRC1);
operand2 ←ZeroExtend32(RSRC2);
IF (operand1 < operand2)
        result ←operand1;
ELSE
        result ←operand2;
```

```
RDEST ←Register(result);
```

Description:

Unsigned minimum

Restrictions:

No address/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)
        result ←operand1;
ELSE
        result ←operand2;
```

```
RIDEST ←Register(result);
```

Description:

Unsigned minimum

Restrictions:

No address/bundle restrictions.

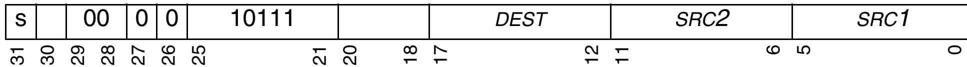
No latency constraints.

Exceptions:

None.

mulh Register

mulh R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend_{32}(R_{SRC1}); operand2 ←SignExtend_{32}(R_{SRC2});
result ←operand1 × (operand2 >> 16); R_{DEST} ←Register(result);

Description:

Word by upper-half-word signed multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mulh Immediate

mulh R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*));
result ←operand1 × (operand2 >> 16); R_{IDEST} ←Register(result);

Description:

Word by upper-half-word signed multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mulhh Register

mulhh $R_{DEST} = R_{SRC1}, R_{SRC2}$

s		00	0	0	11101		DEST	SRC2	SRC1
31	30	29 28	27	26	25 21	20 18	17 12	11 6	5 0

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←SignExtend$_{32}$(R_{SRC2});

result ←(operand1 >> 16) × (operand2 >> 16);
R_{DEST} ←Register(result);

Description:

Upper-half-word by upper-half-word signed multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mulhh Immediate

mulhh R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*));
result ←(operand1 >> 16) × (operand2 >> 16); R_{IDEST} ←Register(result);

Description:

Upper-half-word by upper-half-word signed multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

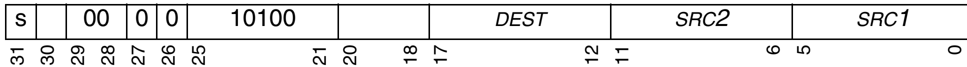
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mulhhs Register

mulhhs $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result ←(operand1 × (operand2 >> 16)) >> 16; R_{DEST} ←Register(result);

Description:

Word by upper-half-word signed multiply, returns top 32 bits of 48 bit result

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mulhhs Immediate

mulhhs R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ← SignExtend$_{32}$(R_{SRC1}); operand2 ← SignExtend$_{32}$(Imm(*ISRC2*));
result ← (operand1 × (operand2 >> 16)) >> 16; R_{IDEST} ← Register(result);

Description:

Word by upper-half-word signed multiply, returns top 32 bits of 48 bit result

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

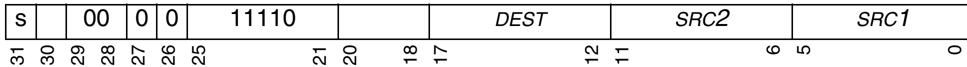
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mulhhu Register

mulhhu $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result ←ZeroExtend$_{16}$(operand1 >> 16) × ZeroExtend$_{16}$(operand2 >> 16); R_{DEST} ←Register(result);

Description:

Upper-half-word by upper-half-word unsigned multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mulhhu Immediate

mulhhu R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*));
result ←ZeroExtend$_{16}$(operand1 >> 16) × ZeroExtend$_{16}$(operand2 >> 16); R_{IDEST} ←Register(result);

Description:

Upper-half-word by upper-half-word unsigned multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

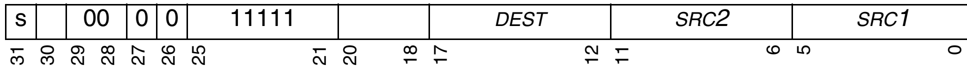
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mulhs Register

mulhs R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result ←(operand1 × ZeroExtend$_{16}$(operand2 >> 16)) << 16; R_{DEST} ←Register(result);

Description:

Word by upper-half-word unsigned multiply, left shifted 16

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mulhs Immediate

mulhs R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*));
result ←(operand1 × ZeroExtend$_{16}$(operand2 >> 16)) << 16; R_{IDEST} ←Register(result);

Description:

Word by upper-half-word unsigned multiply, left shifted 16

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

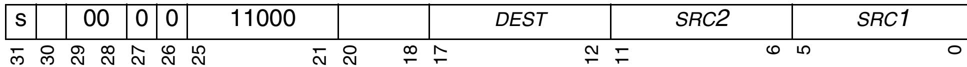
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mulhu Register

mulhu R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result ←operand1 × ZeroExtend$_{16}$(operand2 >> 16); R_{DEST} ←Register(result);

Description:

Word by upper-half-word unsigned multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mulhu Immediate

mulhu R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*));
result ←operand1 × ZeroExtend$_{16}$(operand2 >> 16); R_{IDEST} ←Register(result);

Description:

Word by upper-half-word unsigned multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

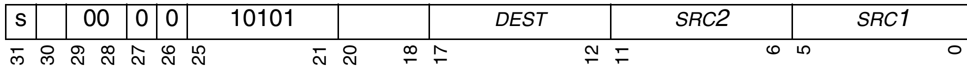
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mull Register

mull R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←$\text{SignExtend}_{32}(R_{SRC1})$; operand2 ←$\text{SignExtend}_{16}(R_{SRC2})$;
result ←operand1 × operand2; R_{DEST} ←Register(result);

Description:

Word by half-word signed multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mull Immediate

mull R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{16}$(Imm(*ISRC2*));
result ←operand1 × operand2; R_{IDEST} ←Register(result);

Description:

Word by half-word signed multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

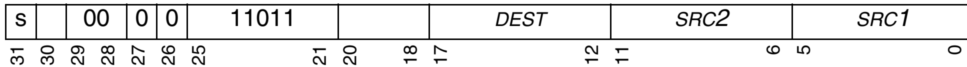
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mullh Register

mullh R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result ←operand1 × (operand2 >> 16); R_{DEST} ←Register(result);

Description:

Half-word by upper-half-word signed multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mullh Immediate

mullh R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*));
result ←operand1 × (operand2 >> 16); R_{IDEST} ←Register(result);

Description:

Half-word by upper-half-word signed multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

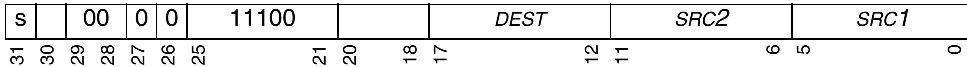
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mullhu Register

mullhu R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2});
result ←operand1 × ZeroExtend$_{16}$(operand2 >> 16); R_{DEST} ←Register(result);

Description:

Half-word by upper-half-word unsigned multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mullhu Immediate

mullhu R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*));
result ←operand1 × ZeroExtend$_{16}$(operand2 >> 16); R_{IDEST} ←Register(result);

Description:

Half-word by upper-half-word unsigned multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

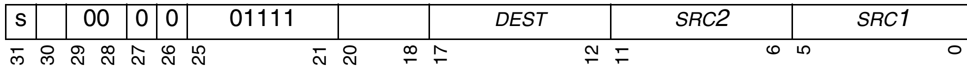
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mullhus Register

mullhus $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
operand2 ←ZeroExtend$_{16}$(R_{SRC2});

result ←(operand1 × operand2) >> 32;
R_{DEST} ←Register(result);

Description:

Word by lower-half-word signed multiply, returns top 16 bits of 48 bit result, sign extended

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mullhus Immediate

mullhus R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{16}$(Imm(*ISRC2*));
result ←(operand1 × operand2) >> 32; R_{IDEST} ←Register(result);

Description:

Word by lower-half-word signed multiply, returns top 16 bits of 48 bit result, sign extended

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

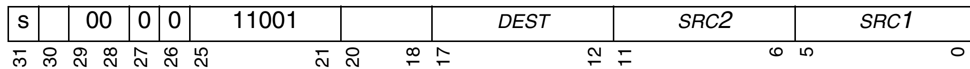
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mulll Register

mulll R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{16}$(R_{SRC2});
result ←operand1 × operand2; R_{DEST} ←Register(result);

Description:

Half-word by half-word signed multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mulll Immediate

mulll R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1}); operand2 ←SignExtend$_{16}$(Imm(*ISRC2*));
result ←operand1 × operand2; R_{IDEST} ←Register(result);

Description:

Half-word by half-word signed multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

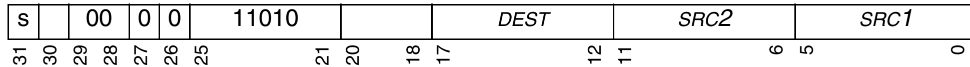
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mulllu Register

mulllu R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1}); operand2 ←ZeroExtend$_{16}$(R_{SRC2});
result ←operand1 × operand2; R_{DEST} ←Register(result);

Description:

Half-word by half-word unsigned multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mulllu Immediate

mulllu R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1}); operand2 ←ZeroExtend$_{16}$(Imm(*ISRC2*));
result ←operand1 × operand2; R_{IDEST} ←Register(result);

Description:

Half-word by half-word unsigned multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

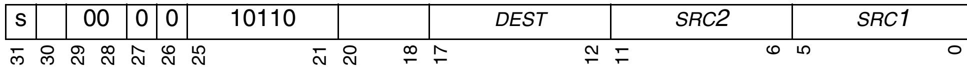
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

mullu Register

mullu $R_{DEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{16}$(R_{SRC2});
result ←operand1 × operand2; R_{DEST} ←Register(result);

Description:

Word by half-word unsigned multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

This instruction must be followed by 2 bundles before R_{DEST} can be read.

Exceptions:

None.

mullu Immediate

mullu R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); operand2 ←ZeroExtend$_{16}$(Imm(*ISRC2*));
result ←operand1 × operand2; R_{IDEST} ←Register(result);

Description:

Word by half-word unsigned multiply

Restrictions:

Cannot write the link register (LR).

Must be encoded at odd word addresses.

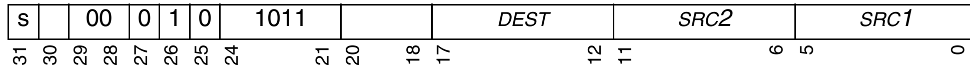
This instruction must be followed by 2 bundles before R_{IDEST} can be read.

Exceptions:

None.

nandl Register - Register

nandl R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←NOT ((operand1 ≠ 0) AND (operand2 ≠ 0));
R_{DEST} ←Register(result);

Description:

Logical nand

Restrictions:

No address/bundle restrictions.

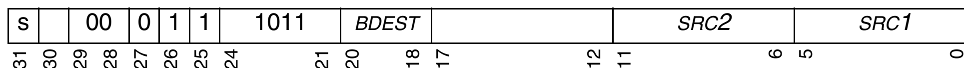
No latency constraints.

Exceptions:

None.

nandl Branch Register - Register

nandl $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←NOT ((operand1 ≠ 0) AND (operand2 ≠ 0));
B_{BDEST} ←Bit(result);

Description:

Logical nand

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

nandl Register - Immediate

nandl R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←NOT ((operand1 ≠ 0) AND (operand2 ≠ 0));
R_{IDEST} ←Register(result);

Description:

Logical nand

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

nandl Branch Register - Immediate

nandl B_{IBDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←NOT ((operand1 ≠ 0) AND (operand2 ≠ 0));
B_{IBDEST} ←Bit(result);

Description:

Logical nand

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

norl Register - Register

norl R_{DEST} = R_{SRC1}, R_{SRC2}

s		00	0	1	0	1101		DEST	SRC2	SRC1
31	30	29 28	27	26	25	24 21	20 18	17 12	11 6	5 0

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←NOT ((operand1 ≠ 0) OR (operand2 ≠ 0));
R_{DEST} ←Register(result);

Description:

Logical nor

Restrictions:

No address/bundle restrictions.

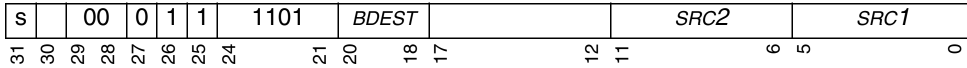
No latency constraints.

Exceptions:

None.

norl Branch Register - Register

norl $B_{BDEST} = R_{SRC1}, R_{SRC2}$



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←NOT ((operand1 ≠ 0) OR (operand2 ≠ 0));
B_{BDEST} ←Bit(result);

Description:

Logical nor

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

norl Register - Immediate

norl R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←NOT ((operand1 ≠ 0) OR (operand2 ≠ 0));
R_{IDEST} ←Register(result);

Description:

Logical nor

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

norl Branch Register - Immediate

norl B_{IBDEST} = R_{SRC1}, *ISRC2*

s		00	1	1	1	1101	*ISRC2*		*IBDEST*	*SRC1*
31	30	29 28	27	26	25	24 21	20 12	11 9	8 6	5 0

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←NOT ((operand1 ≠ 0) OR (operand2 ≠ 0));
B_{IBDEST} ←Bit(result);

Description:

Logical nor

Restrictions:

No address/bundle restrictions.

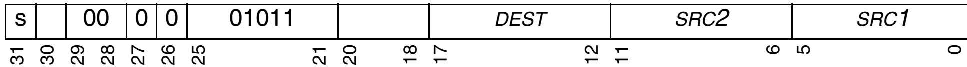
No latency constraints.

Exceptions:

None.

or Register

or R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 ∨ operand2;
R_{DEST} ←Register(result);

Description:

Bitwise or

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

or Immediate

or R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ← $\text{SignExtend}_{32}(R_{SRC1})$; operand2 ← $\text{SignExtend}_{32}(\text{Imm}(ISRC2))$; result ← operand1 ∨ operand2;
R_{IDEST} ← Register(result);

Description:

Bitwise or

Restrictions:

No address/bundle restrictions.

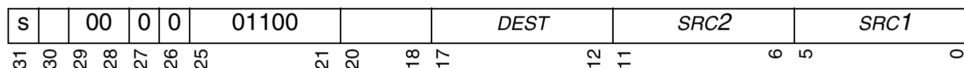
No latency constraints.

Exceptions:

None.

orc Register

orc R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←(~ operand1) ∨ operand2;
R_{DEST} ←Register(result);

Description:

Complement and bitwise or

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

orc Immediate

orc R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ← $SignExtend_{32}(R_{SRC1})$; operand2 ← $SignExtend_{32}(Imm(ISRC2))$; result ← (~ operand1) ∨ operand2;
R_{IDEST} ← Register(result);

Description:

Complement and bitwise or

Restrictions:

No address/bundle restrictions.

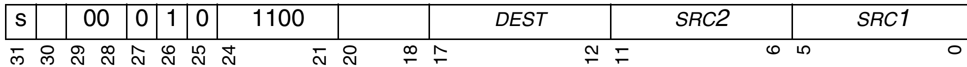
No latency constraints.

Exceptions:

None.

orl Register - Register

orl R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←(operand1 ≠ 0) OR (operand2 ≠ 0);
R_{DEST} ←Register(result);

Description:

Logical or

Restrictions:

No address/bundle restrictions.

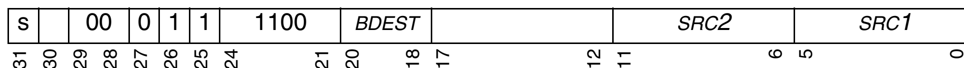
No latency constraints.

Exceptions:

None.

orl Branch Register - Register

orl B_{BDEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←(operand1 ≠ 0) OR (operand2 ≠ 0);
B_{BDEST} ←Bit(result);

Description:

Logical or

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

orl Register - Immediate

orl R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←(operand1 ≠ 0) OR (operand2 ≠ 0);
R_{IDEST} ←Register(result);

Description:

Logical or

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

orl Branch Register - Immediate

orl B_{IBDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←(operand1 ≠ 0) OR (operand2 ≠ 0);
B_{IBDEST} ←Bit(result);

Description:

Logical or

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

pft

pft *ISRC2*[R_{SRC1}]

s		10	01101			*ISRC2*		*SRC1*
31	30	29 28	27 23	22	21	20 12	11 6	5 0

Semantics:

base ←SignExtend$_{32}$(Imm(*ISRC2*)); offset ←SignExtend$_{32}$(R_{SRC1}); ea ←ZeroExtend$_{32}$(base + offset);
PrefetchMemory(ea);

Description:

Prefetch

Restrictions:

Uses the load/store unit, for which only one operation is allowed per bundle.

No latency constraints.

Exceptions:

None.

prgadd

prgadd *ISRC2*[R_{SRC1}]

s		10	01110			*ISRC2*		*SRC1*
31	30	29 28	27 23	22	21	20 12	11 6	5 0

Semantics:

base ←SignExtend$_{32}$(Imm(*ISRC2*)); offset ←SignExtend$_{32}$(R_{SRC1}); ea ←ZeroExtend$_{32}$(base + offset);
PurgeAddress(ea);

Description:

Purge the address given from the data cache

Restrictions:

Uses the load/store unit, for which only one operation is allowed per bundle.

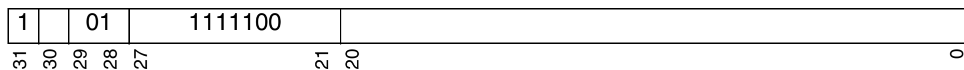
No latency constraints.

Exceptions:

None.

prgins

prgins



Semantics:

PurgeIns();

Description:

Purge the instruction cache

Restrictions:

Must be the only one operation in the bundle.

Must be followed by 3 bundles delay before issuing a syncins operation.

No latency constraints.

Exceptions:

None.

prgset

prgset *ISRC2*[R_{SRC1}]

s		10	01111			*ISRC2*		*SRC1*
31	30	29 28	27 23	22	21	20 12	11 6	5 0

Semantics:

base ←SignExtend$_{32}$(Imm(*ISRC2*)); offset ←SignExtend$_{32}$(R_{SRC1}); ea ←ZeroExtend$_{32}$(base + offset);
PurgeSet(ea);

Description:

Purge a set of four cache lines from the data cache

Restrictions:

Uses the load/store unit, for which only one operation is allowed per bundle.

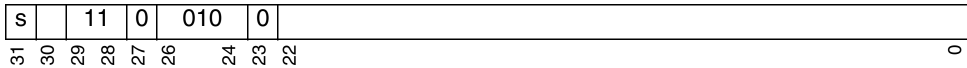
No latency constraints.

Exceptions:

None.

rfi

rfi



Semantics:

```
IF (PSW[USER_MODE])
        THROW ILL_INST;
PC ←Register(ZeroExtend32(SAVED_PC));
```

```
PSW ←SAVED_PSW;
SAVED_PC ←SAVED_SAVED_PC;
SAVED_PSW ←SAVED_SAVED_PSW;
```

Description:

Return from interrupt

Restrictions:

Must be the first syllable of a bundle.

Instructions writing SAVED_PC must be followed by 4 bundles before this instruction can be issued.

Instructions writing SAVED_PSW must be followed by 4 bundles before this instruction can be issued.

Instructions writing SAVED_SAVED_PC must be followed by 4 bundles before this instruction can be issued.

Instructions writing SAVED_SAVED_PSW must be followed by 4 bundles before this instruction can be issued.

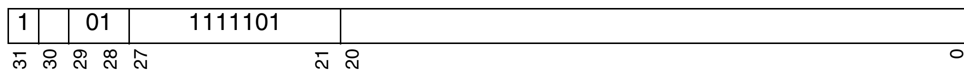
Instructions writing PSW must be followed by 4 bundles before this instruction can be issued.

Exceptions:

ILL_INST

sbrk

sbrk



Semantics:

THROW SBREAK;

Description:

Software breakpoint

Restrictions:

No address/bundle restrictions.

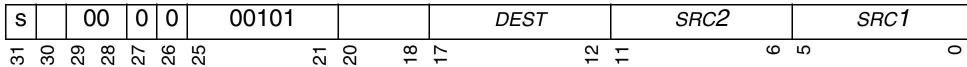
No latency constraints.

Exceptions:

SBREAK

sh1add Register

sh1add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ← $\text{SignExtend}_{32}(R_{SRC1})$; operand2 ← $\text{SignExtend}_{32}(R_{SRC2})$; result ←(operand1 << 1) + operand2;
R_{DEST} ←Register(result);

Description:

Shift left one and accumulate

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

sh1add Immediate

sh1add R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←(operand1 << 1) + operand2;
R_{IDEST} ←Register(result);

Description:

Shift left one and accumulate

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

sh2add Register

sh2add $R_{DEST} = R_{SRC1}, R_{SRC2}$

31	30	29-28	27	26	25-21	20-18	17-12	11-6	5-0
s		00	0	0	00110		*DEST*	*SRC2*	*SRC1*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←(operand1 << 2) + operand2;
R_{DEST} ←Register(result);

Description:

Shift left two and accumulate

Restrictions:

No address/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*)); result ←(operand1 << 2) + operand2;
R_{IDEST} ←Register(result);

Description:

Shift left two and accumulate

Restrictions:

No address/bundle restrictions.

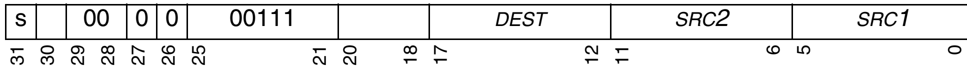
No latency constraints.

Exceptions:

None.

sh3add Register

sh3add R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←(operand1 << 3) + operand2;
R_{DEST} ←Register(result);

Description:

Shift left three and accumulate

Restrictions:

No address/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*)); result ←(operand1 << 3) + operand2;
R_{IDEST} ←Register(result);

Description:

Shift left three and accumulate

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

sh4add Register

sh4add R_{DEST} = R_{SRC1}, R_{SRC2}

s		00	0	0	01000		DEST	SRC2	SRC1
31	30	29 28	27	26	25 21	20 18	17 12	11 6	5 0

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←(operand1 << 4) + operand2;
R_{DEST} ←Register(result);

Description:

Shift left four and accumulate

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

sh4add Immediate

sh4add R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←(operand1 << 4) + operand2;
R_{IDEST} ←Register(result);

Description:

Shift left four and accumulate

Restrictions:

No address/bundle restrictions.

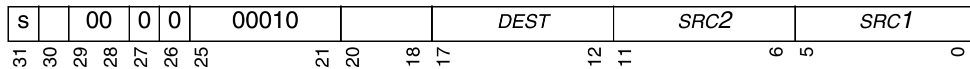
No latency constraints.

Exceptions:

None.

shl Register

shl R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1});
distance ←ZeroExtend$_{8}$(R_{SRC2});
IF (distance > 31)
    result ←0;
ELSE
    result ←operand1 << distance;

R_{DEST} ←Register(result);

Description:

Shift left

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

shl Immediate

shl R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

```
operand1 ←SignExtend32(R_SRC1);
distance ←ZeroExtend8(Imm(ISRC2));
IF (distance > 31)
        result ←0;
ELSE
        result ←operand1 << distance;
```

```
R_IDEST ←Register(result);
```

Description:

Shift left

Restrictions:

No address/bundle restrictions.

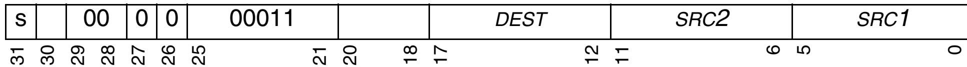
No latency constraints.

Exceptions:

None.

shr Register

shr R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←$SignExtend_{32}(R_{SRC1})$; distance ←$ZeroExtend_{8}(R_{SRC2})$; result ←operand1 >> distance;
R_{DEST} ←Register(result);

Description:

Arithmetic shift right

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

shr Immediate

shr R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); distance ←ZeroExtend$_{8}$(Imm(*ISRC2*)); result ←operand1 >> distance;
R_{IDEST} ←Register(result);

Description:

Arithmetic shift right

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

shru Register

shru $R_{DEST} = R_{SRC1}, R_{SRC2}$

s		00	0	0	00100		DEST	SRC2	SRC1
31	30	29 28	27	26	25 21	20 18	17 12	11 6	5 0

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1});
distance ←ZeroExtend$_{8}$(R_{SRC2});
result ←operand1 >> distance;

R_{DEST} ←Register(result);

Description:

Logical shift right

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

shru Immediate

shru R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←ZeroExtend$_{32}$(R_{SRC1}); distance ←ZeroExtend$_8$(Imm(*ISRC2*)); result ←operand1 >> distance;
R_{IDEST} ←Register(result);

Description:

Logical shift right

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

slct Register

slct R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}

s		01	0	000	*SCOND*		*DEST*	*SRC2*	*SRC1*
31	30	29 28	27	26 24	23 21	20 18	17 12	11 6	5 0

Semantics:

```
operand1 ←ZeroExtend1(B_SCOND);
operand2 ←SignExtend32(R_SRC1);
operand3 ←SignExtend32(R_SRC2);
IF (operand1 ≠ 0)
        result ←operand2;
ELSE
        result ←operand3;
```

```
R_DEST ←Register(result);
```

Description:

Conditional select

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

slct Immediate

slct R_{IDEST} = B_{SCOND}, R_{SRC1}, *ISRC2*

Semantics:

```
operand1 ←ZeroExtend1(B_SCOND);
operand2 ←SignExtend32(R_SRC1);
operand3 ←SignExtend32(Imm(ISRC2));
IF (operand1 ≠ 0)
        result ←operand2;
ELSE
        result ←operand3;
```

```
R_IDEST ←Register(result);
```

Description:

Conditional select

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

slctf Register

slctf R_{DEST} = B_{SCOND}, R_{SRC1}, R_{SRC2}

s		01	0	001	*SCOND*		*DEST*	*SRC2*	*SRC1*
31	30	29 28	27	26 24	23 21	20 18	17 12	11 6	5 0

Semantics:

```
operand1 ←ZeroExtend1(BSCOND);
operand2 ←SignExtend32(RSRC1);
operand3 ←SignExtend32(RSRC2);
IF (operand1 = 0)
        result ←operand2;
ELSE
        result ←operand3;
```

```
RDEST ←Register(result);
```

Description:

Conditional select

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

slctf Immediate

slctf R_{IDEST} = B_{SCOND}, R_{SRC1}, *ISRC2*

Semantics:

```
operand1 ←ZeroExtend1(BSCOND);
operand2 ←SignExtend32(RSRC1);
operand3 ←SignExtend32(Imm(ISRC2));
IF (operand1 = 0)
        result ←operand2;
ELSE
        result ←operand3;
```

```
RIDEST ←Register(result);
```

Description:

Conditional select

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

stb

stb *ISRC2*[R_{SRC1}] = R_{SRC2}

s		10	01100			*ISRC2*	*SRC2*	*SRC1*
31	30	29 28	27 23	22	21	20 12	11 6	5 0

Semantics:

base ← $SignExtend_{32}$(Imm(*ISRC2*));
offset ← $SignExtend_{32}$(R_{SRC1});
operand3 ← $SignExtend_{32}$(R_{SRC2});
ea ← $ZeroExtend_{32}$(base + offset);
IF (IsDBreakHit(ea))
    THROW DBREAK;
IF (IsCRegSpace(ea))
    THROW CREG_ACCESS_VIOLATION;
$WriteCheckMemory_{8}$(ea);

$WriteMemory_{8}$(ea, operand3);

Description:

Store byte

Restrictions:

Uses the load/store unit, for which only one operation is allowed per bundle.

No latency constraints.

Exceptions:

DBREAK
CREG_ACCESS_VIOLATION
DBREAK
MISALIGNED_TRAP
DPU_NO_TRANSLATION
DPU_ACCESS_VIOLATION

sth

sth *ISRC2*[R_{SRC1}] = R_{SRC2}

s		10	01011			*ISRC2*	*SRC2*	*SRC1*
31	30	29 28	27 23	22	21	20 12	11 6	5 0

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
operand3 ←SignExtend32(RSRC2);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOLATION;
WriteCheckMemory16(ea);
```

```
WriteMemory16(ea, operand3);
```

Description:

Store half-word

Restrictions:

Uses the load/store unit, for which only one operation is allowed per bundle.

No latency constraints.

Exceptions:

DBREAK

CREG_ACCESS_VIOLATION

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

stw

stw *ISRC2*[R_{SRC1}] = R_{SRC2}

Semantics:

```
base ←SignExtend32(Imm(ISRC2));
offset ←SignExtend32(RSRC1);
operand3 ←SignExtend32(RSRC2);
ea ←ZeroExtend32(base + offset);
IF (IsDBreakHit(ea))
        THROW DBREAK;
IF (IsCRegSpace(ea))
        WriteCheckCReg(ea);
WriteCheckMemory32(ea);
```

```
IF (IsCRegSpace(ea))
        WriteCReg(ea, operand3);
ELSE
        WriteMemory32(ea, operand3);
```

Description:

Store word

Restrictions:

Uses the load/store unit, for which only one operation is allowed per bundle.

No latency constraints.

Exceptions:

DBREAK

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

CREG_NO_MAPPING

CREG_ACCESS_VIOLATION

sub Register

sub R_{DEST} = R_{SRC2}, R_{SRC1}

s		00	0	0	00001		*DEST*	*SRC2*	*SRC1*
31	30	29 28	27	26	25 21	20 18	17 12	11 6	5 0

Semantics:

operand2 ← $SignExtend_{32}(R_{SRC2})$; operand1 ← $SignExtend_{32}(R_{SRC1})$; result ←operand2 - operand1;
R_{DEST} ←Register(result);

Description:

Subtract

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

sub Immediate

sub R_{IDEST} = *ISRC2*, R_{SRC1}

Semantics:

operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); operand1 ←SignExtend$_{32}$(R_{SRC1}); result ←operand2 - operand1;
R_{IDEST} ←Register(result);

Description:

Subtract

Restrictions:

No address/bundle restrictions.

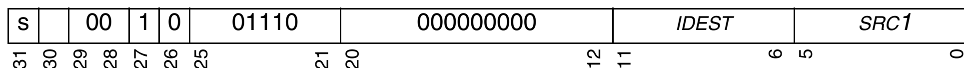
No latency constraints.

Exceptions:

None.

sxtb

sxtb R_{IDEST} = R_{SRC1}



Semantics:

operand1 ←SignExtend$_8$(R_{SRC1}); result ←operand1;
R_{IDEST} ←Register(result);

Description:

Sign extend byte

Restrictions:

No address/bundle restrictions.

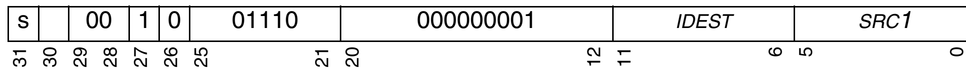
No latency constraints.

Exceptions:

None.

sxth

sxth R_{IDEST} = R_{SRC1}



Semantics:

operand1 ←SignExtend$_{16}$(R_{SRC1}); result ←operand1;
R_{IDEST} ←Register(result);

Description:

Sign extend half

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

sync

sync

1		10	10000					
31	30	29 28	27 23	22	21	20 12	11 6	5 0

Semantics:

Sync();

Description:

Ensure synchronisation

Restrictions:

Uses the load/store unit, for which only one operation is allowed per bundle.

Must be the only syllable in the bundle.

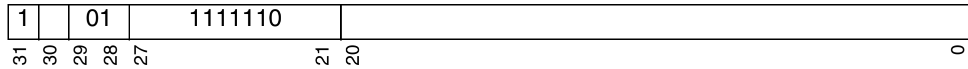
No latency constraints.

Exceptions:

None.

syscall

syscall



Semantics:

THROW ILL_INST;

Description:

System call

Restrictions:

Must be the only syllable in the bundle.

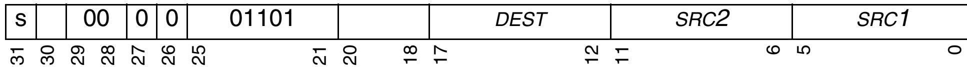
No latency constraints.

Exceptions:

ILL_INST

xor Register

xor R_{DEST} = R_{SRC1}, R_{SRC2}



Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(R_{SRC2}); result ←operand1 ⊕ operand2;
R_{DEST} ←Register(result);

Description:

Bitwise exclusive-or

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

xor Immediate

xor R_{IDEST} = R_{SRC1}, *ISRC2*

Semantics:

operand1 ←SignExtend$_{32}$(R_{SRC1}); operand2 ←SignExtend$_{32}$(Imm(*ISRC2*)); result ←operand1 ⊕ operand2;
R_{IDEST} ←Register(result);

Description:

Bitwise exclusive-or

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

zxth

zxth R_{IDEST} = R_{SRC1}

s		00	1	0	01110	000000011	*IDEST*	*SRC1*
31	30	29 28	27	26	25 21	20 12	11 6	5 0

Semantics:

operand1 ←ZeroExtend$_{16}$(R_{SRC1}); result ←operand1;
R_{IDEST} ←Register(result);

Description:

Zero extend half

Restrictions:

No address/bundle restrictions.

No latency constraints.

Exceptions:

None.

Instruction encoding

A

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 the following table.

Field Name	Offset	Size	Field represents
BCOND	23	3	Branch register containing the branch condition.
BDEST	18	3	Destination branch register for register format operations.
BTARG	0	23	Branch offset value from PC.
DEST	12	6	Destination general purpose register for register format operations.
IBDEST	6	3	Destination branch register for immediate format operations.
ICBUS	12	9	Intercluster bus.
IDEST	6	6	Destination general purpose register for immediate format operations.
IMM	0	23	23-bit value used to extend a short immediate.
ISRC2	12	9	9-bit short immediate value.
SCOND	21	3	Source branch register used for select condition or carry.
SRC1	0	6	General purpose source register.
SRC2	6	6	General purpose source register.

Figure 17: Operand fields

A.3 Formats

<table>
<tr><th></th><th>Bundle Stop</th><th>Cluster Bit</th><th colspan="2">Format</th><th colspan="7">Opcode</th><th colspan="9">Immediate/ Dest</th><th colspan="6">Dest/Src2</th><th colspan="6">Src1</th></tr>
<tr><th></th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Int3R</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">Opcode</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>Int3I</td><td>s</td><td></td><td colspan="2">00</td><td>1</td><td>0</td><td colspan="5">Opcode</td><td colspan="9">ISRC2</td><td colspan="6">IDEST</td><td colspan="6">SRC1</td></tr>
<tr><td>Monadic</td><td>s</td><td></td><td colspan="2">00</td><td>1</td><td>0</td><td colspan="5">01110</td><td colspan="9">Opcode</td><td colspan="6">IDEST</td><td colspan="6">SRC1</td></tr>
<tr><td>Cmp3R_Reg</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>1</td><td>0</td><td colspan="4">Opcode</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>Cmp3R_Br</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>1</td><td>1</td><td colspan="4">Opcode</td><td colspan="3">BDEST</td><td colspan="6"></td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>Cmp3I_Reg</td><td>s</td><td></td><td colspan="2">00</td><td>1</td><td>1</td><td>0</td><td colspan="4">Opcode</td><td colspan="9">ISRC2</td><td colspan="6">IDEST</td><td colspan="6">SRC1</td></tr>
<tr><td>Cmp3I_Br</td><td>s</td><td></td><td colspan="2">00</td><td>1</td><td>1</td><td>1</td><td colspan="4">Opcode</td><td colspan="9">ISRC2</td><td colspan="3"></td><td colspan="3">IBDEST</td><td colspan="6">SRC1</td></tr>
<tr><td>Imm</td><td>s</td><td></td><td colspan="2">01</td><td colspan="5">Opcode</td><td colspan="23">IMM</td></tr>
<tr><td>SelectR</td><td>s</td><td></td><td colspan="2">01</td><td>0</td><td colspan="3">Opcode</td><td colspan="3">SCOND</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>SelectI</td><td>s</td><td></td><td colspan="2">01</td><td>1</td><td colspan="3">Opcode</td><td colspan="3">SCOND</td><td colspan="9">ISRC2</td><td colspan="6">IDEST</td><td colspan="6">SRC1</td></tr>
<tr><td>cgen</td><td>s</td><td></td><td colspan="2">01</td><td colspan="4">Opcode</td><td colspan="3">SCOND</td><td colspan="3">BDEST</td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>SysOp</td><td>s</td><td></td><td colspan="2">01</td><td colspan="7">Opcode</td><td colspan="21"></td></tr>
<tr><td>Load</td><td>s</td><td></td><td colspan="2">10</td><td colspan="4">Opcode</td><td></td><td></td><td></td><td colspan="9">ISRC2</td><td colspan="6">IDEST</td><td colspan="6">SRC1</td></tr>
<tr><td>Store</td><td>s</td><td></td><td colspan="2">10</td><td colspan="5">Opcode</td><td></td><td></td><td colspan="9">ISRC2</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>Call</td><td>s</td><td></td><td colspan="2">11</td><td>0</td><td colspan="3">Opcode</td><td></td><td colspan="23">BTARG</td></tr>
<tr><td>Branch</td><td>s</td><td></td><td colspan="2">11</td><td>1</td><td>Opcode</td><td colspan="3">BCOND</td><td colspan="23">BTARG</td></tr>
</table>

Figure 18: Formats

Several important points:

- The BUNDLE STOP bit indicates the end of bundle and is set in the last syllable of the bundle.
- The CLUSTER bit is reserved; in ST220 it is set to zero.
- 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** R_{SRC2}. 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 (B_{BDEST}). 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.4 Opcodes

* These operations are not supported by the hardware, but the opcodes are reserved for software investigations.

<table>
<tr><th></th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>add</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">00000</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>sub</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">00001</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>shl</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">00010</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>shr</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">00011</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>shru</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">00100</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>sh1add</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">00101</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>sh2add</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">00110</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>sh3add</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">00111</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>sh4add</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">01000</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>and</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">01001</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>andc</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">01010</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>or</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">01011</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>orc</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">01100</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>xor</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">01101</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mullhus</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">01111</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>max</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">10000</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>maxu</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">10001</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>min</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">10010</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>minu</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">10011</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mulhhs</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">10100</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mull</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">10101</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mullu</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">10110</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mulh</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">10111</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mulhu</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">11000</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mulll</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">11001</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mulllu</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">11010</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mullh</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">11011</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mullhu</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">11100</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mulhh</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">11101</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mulhhu</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">11110</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>mulhs</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>0</td><td colspan="5">11111</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
<tr><td>cmpeq</td><td>s</td><td></td><td colspan="2">00</td><td>0</td><td>1</td><td>0</td><td colspan="4">0000</td><td colspan="3"></td><td colspan="6">DEST</td><td colspan="6">SRC2</td><td colspan="6">SRC1</td></tr>
</table>

Figure 19: Instruction encodings

	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
cmpne	s		00	0	1	0	0001		*DEST*	*SRC2*	*SRC1*
cmpge	s		00	0	1	0	0010		*DEST*	*SRC2*	*SRC1*
cmpgeu	s		00	0	1	0	0011		*DEST*	*SRC2*	*SRC1*
cmpgt	s		00	0	1	0	0100		*DEST*	*SRC2*	*SRC1*
cmpgtu	s		00	0	1	0	0101		*DEST*	*SRC2*	*SRC1*
cmple	s		00	0	1	0	0110		*DEST*	*SRC2*	*SRC1*
cmpleu	s		00	0	1	0	0111		*DEST*	*SRC2*	*SRC1*
cmplt	s		00	0	1	0	1000		*DEST*	*SRC2*	*SRC1*
cmpltu	s		00	0	1	0	1001		*DEST*	*SRC2*	*SRC1*
andl	s		00	0	1	0	1010		*DEST*	*SRC2*	*SRC1*
nandl	s		00	0	1	0	1011		*DEST*	*SRC2*	*SRC1*
orl	s		00	0	1	0	1100		*DEST*	*SRC2*	*SRC1*
norl	s		00	0	1	0	1101		*DEST*	*SRC2*	*SRC1*
cmpeq	s		00	0	1	1	0000	*BDEST*		*SRC2*	*SRC1*
cmpne	s		00	0	1	1	0001	*BDEST*		*SRC2*	*SRC1*
cmpge	s		00	0	1	1	0010	*BDEST*		*SRC2*	*SRC1*
cmpgeu	s		00	0	1	1	0011	*BDEST*		*SRC2*	*SRC1*
cmpgt	s		00	0	1	1	0100	*BDEST*		*SRC2*	*SRC1*
cmpgtu	s		00	0	1	1	0101	*BDEST*		*SRC2*	*SRC1*
cmple	s		00	0	1	1	0110	*BDEST*		*SRC2*	*SRC1*
cmpleu	s		00	0	1	1	0111	*BDEST*		*SRC2*	*SRC1*
cmplt	s		00	0	1	1	1000	*BDEST*		*SRC2*	*SRC1*
cmpltu	s		00	0	1	1	1001	*BDEST*		*SRC2*	*SRC1*
andl	s		00	0	1	1	1010	*BDEST*		*SRC2*	*SRC1*
nandl	s		00	0	1	1	1011	*BDEST*		*SRC2*	*SRC1*
orl	s		00	0	1	1	1100	*BDEST*		*SRC2*	*SRC1*
norl	s		00	0	1	1	1101	*BDEST*		*SRC2*	*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*
sh4add	s		00	1	0	01000		*ISRC2*		*IDEST*	*SRC1*
and	s		00	1	0	01001		*ISRC2*		*IDEST*	*SRC1*

Figure 19: Instruction encodings

	31	30	29–28	27	26	25	24–21	20–12	11–6	5–0
andc	s		00	1	0	01010		*ISRC2*	*IDEST*	*SRC1*
or	s		00	1	0	01011		*ISRC2*	*IDEST*	*SRC1*
orc	s		00	1	0	01100		*ISRC2*	*IDEST*	*SRC1*
xor	s		00	1	0	01101		*ISRC2*	*IDEST*	*SRC1*
sxtb	s		00	1	0	01110		000000000	*IDEST*	*SRC1*
sxth	s		00	1	0	01110		000000001	*IDEST*	*SRC1*
bswap	s		00	1	0	01110		000000010	*IDEST*	*SRC1*
zxth	s		00	1	0	01110		000000011	*IDEST*	*SRC1*
clz	s		00	1	0	01110		000000100	*IDEST*	*SRC1*
mullhus	s		00	1	0	01111		*ISRC2*	*IDEST*	*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*
mulhhs	s		00	1	0	10100		*ISRC2*	*IDEST*	*SRC1*
mull	s		00	1	0	10101		*ISRC2*	*IDEST*	*SRC1*
mullu	s		00	1	0	10110		*ISRC2*	*IDEST*	*SRC1*
mulh	s		00	1	0	10111		*ISRC2*	*IDEST*	*SRC1*
mulhu	s		00	1	0	11000		*ISRC2*	*IDEST*	*SRC1*
mulll	s		00	1	0	11001		*ISRC2*	*IDEST*	*SRC1*
mulllu	s		00	1	0	11010		*ISRC2*	*IDEST*	*SRC1*
mullh	s		00	1	0	11011		*ISRC2*	*IDEST*	*SRC1*
mullhu	s		00	1	0	11100		*ISRC2*	*IDEST*	*SRC1*
mulhh	s		00	1	0	11101		*ISRC2*	*IDEST*	*SRC1*
mulhhu	s		00	1	0	11110		*ISRC2*	*IDEST*	*SRC1*
mulhs	s		00	1	0	11111		*ISRC2*	*IDEST*	*SRC1*
cmpeq	s		00	1	1	0	0000	*ISRC2*	*IDEST*	*SRC1*
cmpne	s		00	1	1	0	0001	*ISRC2*	*IDEST*	*SRC1*
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*
andl	s		00	1	1	0	1010	*ISRC2*	*IDEST*	*SRC1*

Figure 19: Instruction encodings

	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
nandl	s		00		1	1	0	1011				*ISRC2*									*IDEST*						*SRC1*					
orl	s		00		1	1	0	1100				*ISRC2*									*IDEST*						*SRC1*					
norl	s		00		1	1	0	1101				*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*					
andl	s		00		1	1	1	1010				*ISRC2*												*IBDEST*			*SRC1*					
nandl	s		00		1	1	1	1011				*ISRC2*												*IBDEST*			*SRC1*					
orl	s		00		1	1	1	1100				*ISRC2*												*IBDEST*			*SRC1*					
norl	s		00		1	1	1	1101				*ISRC2*												*IBDEST*			*SRC1*					
slct	s		01		0	000			*SCOND*						*DEST*						*SRC2*						*SRC1*					
slctf	s		01		0	001			*SCOND*						*DEST*						*SRC2*						*SRC1*					
addcg	s		01		0010				*SCOND*			*BDEST*			*DEST*						*SRC2*						*SRC1*					
divs	s		01		0100				*SCOND*			*BDEST*			*DEST*						*SRC2*						*SRC1*					
imml	s		01		01010					*IMM*																						
immr	s		01		01011					*IMM*																						
slct	s		01		1	000			*SCOND*			*ISRC2*									*IDEST*						*SRC1*					
slctf	s		01		1	001			*SCOND*			*ISRC2*									*IDEST*						*SRC1*					
prgins	1		01		1111100																											
sbrk	1		01		1111101																											
syscall	1		01		1111110																											
break	s		01		1111111																											
ldw	s		10		0000				0			*ISRC2*									*IDEST*						*SRC1*					
ldw.d	s		10		0000				1			*ISRC2*									*IDEST*						*SRC1*					
ldh	s		10		0001				0			*ISRC2*									*IDEST*						*SRC1*					
ldh.d	s		10		0001				1			*ISRC2*									*IDEST*						*SRC1*					
ldhu	s		10		0010				0			*ISRC2*									*IDEST*						*SRC1*					
ldhu.d	s		10		0010				1			*ISRC2*									*IDEST*						*SRC1*					
ldb	s		10		0011				0			*ISRC2*									*IDEST*						*SRC1*					
ldb.d	s		10		0011				1			*ISRC2*									*IDEST*						*SRC1*					

Figure 19: Instruction encodings

	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
ldbu	s		10	0100				0			*ISRC2*		*IDEST*	*SRC1*
ldbu.d	s		10	0100				1			*ISRC2*		*IDEST*	*SRC1*
stw	s		10	01010							*ISRC2*		*SRC2*	*SRC1*
sth	s		10	01011							*ISRC2*		*SRC2*	*SRC1*
stb	s		10	01100							*ISRC2*		*SRC2*	*SRC1*
pft	s		10	01101							*ISRC2*			*SRC1*
prgadd	s		10	01110							*ISRC2*			*SRC1*
prgset	s		10	01111							*ISRC2*			*SRC1*
sync	1		10	10000										
send *	s		10	1	0	00100					*ICBUS*		*SRC2*	
recv *	s		10	1	0	01000					*ICBUS*		*IDEST*	
asm,0 *	s		10	1	1	00000						*DEST*	*SRC2*	*SRC1*
asm,1 *	s		10	1	1	00001						*DEST*	*SRC2*	*SRC1*
asm,2 *	s		10	1	1	00010						*DEST*	*SRC2*	*SRC1*
asm,3 *	s		10	1	1	00011						*DEST*	*SRC2*	*SRC1*
asm,4 *	s		10	1	1	00100						*DEST*	*SRC2*	*SRC1*
asm,5 *	s		10	1	1	00101						*DEST*	*SRC2*	*SRC1*
asm,6 *	s		10	1	1	00110						*DEST*	*SRC2*	*SRC1*
asm,7 *	s		10	1	1	00111						*DEST*	*SRC2*	*SRC1*
asm,8 *	s		10	1	1	01000						*DEST*	*SRC2*	*SRC1*
asm,9 *	s		10	1	1	01001						*DEST*	*SRC2*	*SRC1*
asm,10 *	s		10	1	1	01010						*DEST*	*SRC2*	*SRC1*
asm,11 *	s		10	1	1	01011						*DEST*	*SRC2*	*SRC1*
asm,12 *	s		10	1	1	01100						*DEST*	*SRC2*	*SRC1*
asm,13 *	s		10	1	1	01101						*DEST*	*SRC2*	*SRC1*
asm,14 *	s		10	1	1	01110						*DEST*	*SRC2*	*SRC1*
asm,15 *	s		10	1	1	01111						*DEST*	*SRC2*	*SRC1*
asm,0 *	s		10	1	1	10000					*ISRC2*		*IDEST*	*SRC1*
asm,1 *	s		10	1	1	10001					*ISRC2*		*IDEST*	*SRC1*
asm,2 *	s		10	1	1	10010					*ISRC2*		*IDEST*	*SRC1*
asm,3 *	s		10	1	1	10011					*ISRC2*		*IDEST*	*SRC1*
asm,4 *	s		10	1	1	10100					*ISRC2*		*IDEST*	*SRC1*
asm,5 *	s		10	1	1	10101					*ISRC2*		*IDEST*	*SRC1*
asm,6 *	s		10	1	1	10110					*ISRC2*		*IDEST*	*SRC1*
asm,7 *	s		10	1	1	10111					*ISRC2*		*IDEST*	*SRC1*
asm,8 *	s		10	1	1	11000					*ISRC2*		*IDEST*	*SRC1*
asm,9 *	s		10	1	1	11001					*ISRC2*		*IDEST*	*SRC1*

Figure 19: Instruction encodings

	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
asm,10 *	s		10		1	1	11010					*ISRC2*									*IDEST*						*SRC1*					
asm,11 *	s		10		1	1	11011					*ISRC2*									*IDEST*						*SRC1*					
asm,12 *	s		10		1	1	11100					*ISRC2*									*IDEST*						*SRC1*					
asm,13 *	s		10		1	1	11101					*ISRC2*									*IDEST*						*SRC1*					
asm,14 *	s		10		1	1	11110					*ISRC2*									*IDEST*						*SRC1*					
asm,15 *	s		10		1	1	11111					*ISRC2*									*IDEST*						*SRC1*					
call	s		11		0	000			0	*BTARG*																						
call	s		11		0	000			1																							
goto	s		11		0	001			0	*BTARG*																						
goto	s		11		0	001			1																							
rfi	s		11		0	010			0																							
br	s		11		1	0	*BCOND*			*BTARG*																						
brf	s		11		1	1	*BCOND*			*BTARG*																						

Figure 19: Instruction encodings

Glossary

B

Branch registers The set of eight 1-bit registers that encode the condition for conditional branches and carry bits.

Bundle Wide instruction of multiple operations always issued during the same cycle and executed in parallel.

Cluster Collection of tightly coupled functional units and register files that perform the computational tasks in an ST200.

Control register One of a set of address mapped registers maintained by the hardware (or operating system or user). These registers may have side effects and may require supervisor access permissions.

Dispersal The operation of extracting and routing the syllables of one bundle stored in the I-cache line to the proper slots in the bundle buffer. To avoid using I-cache space for empty syllables, only non-empty syllables are stored in the I-cache lines. Hence, it is necessary to “disperse” the syllable of each bundle to the full-size bundle buffer.

General-purpose registers The set of directly addressed fixed-point registers. ST200 contains one GR file per cluster organized as a bank of 64 32-bit registers. The compiler is responsible for explicitly scheduling data transfers among GRs residing in different clusters.

Level-1 I-cache Level-1 instruction cache also referred as the “closest” or “lowest” cache. Similar notations apply to the Level-1 data cache. ST200 supports multiple Level-1 data caches.

Main memory This is the system-accessible memory, cached.

Operation An operation is an atomic ST200 action, in general considered roughly 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). One of the simplest forms of predication is a *select* operation, which is supported in ST200.

Speculative A speculative operation (also known as “eager”) is an operation executed prior to the resolution of the branch under which the operation would normally execute. Special attention must be paid to speculative memory load operations to handle the possible resulting exceptions. Speculative memory load operation are sometimes called “dismissible” as any exception deriving from the operation has to be ignored (“dismissed”) by the system.

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 to be executed by the machine functional units. Syllables are composed of register and/or immediate fields and opcode specifiers. A bundle in ST200 may contain multiple syllables, each of them 32-bit wide. The syllable is also the indivisible unit for the bundle compression/dispersal.

VLIW Very long instruction word: instructions (called “bundles” in ST200 terminology) potentially encode multiple, independent operations, and are fully scheduled at compile time.

Index

Numerics

R 25

A

AND 116
architecture 324

B

B 123
branch 324
bundle 323-324

C

cache 323
carry 323
cluster 323
Commit point 105, 137
compiler 323
compression 324
conditional 323
CR 123

D

dismissible 324
dispersal 324
DPU 45

E

ELSE 120
exception 324

F

FOR 114-115, 117, 121, 125, 127-128, 130
FROM 121
Function
- Bit(i) 117
- BusReadError(address) 124
- Commit(n) 108
- ControlRegister(address) 124
- CregReadAccessViolation(index) 124
- CregWriteAccessViolation(index) 124
- DataBreakPoint(address) 124
- DisReadCheckMemory(address) 126
- DisReadMemory(address) 126
- DPUNoTranslation(address) 124
- DPUSpecLoadRetZero(address) 124
- Imm(i) 136
- InitiateDebugIntHandler() 108
- InitiateExceptionHandler() 108
- IsControlSpace(address) 124
- IsDBreakHit(address) 124
- Misaligned(address) 124
- NumExtImms(address) 108

NumWords(address) 108
Pre-commit(n) 108
Prefetch(address) 133
PrefetchMemory(address) 129
PurgeAddress(address) 133
PurgeIns( ) 133
PurgeSet(address) 133
ReadAccessViolation(address) 124
ReadCheckControl(address) 131
ReadCheckMemory(address) 126
ReadControl(address) 131
ReadMemory(address) 126
Register(i) 117
SignExtend(i) 117
Sync( ) 133
UndefinedControlRegister(address) 124
WriteAccessViolation(address) 124
WriteCheckControl(address) 132
WriteCheckMemory(address) 130
WriteControl(address, value) 132
WriteMemory(address, value) 130
ZeroExtend(i) 117

I

IF 120, 127-128
immediate 324
instruction 323-324
INT 116
integer 13
IPU 45

L

load 324
LR 122

M

MEM 123, 125, 127-128, 130

N

NOT 116

O

operations 323-324
OR 116
or 323-324
Overlapping regions 41

P

parallel 323
PC 25, 122
PSW 122

R

R 122
register 323-324
REPEAT 121

S

SAVED_PC 122
SAVED_PSW 122
SAVED_SAVED_PC 122
SAVED_SAVED_PSW 122
select 324
speculation 324
STEP 121
stop bit 108, 135
syllable 323-324

T

THROW 121, 127-128

U

UNDEFINED 118-119
Undefined address space 42

XYZ

XOR 116