()
TITLE: ST220 INSTRUCTION SET ARCHITECTURE
TYPE : PRD−RDI (PRODUCT SPECIFICATION,BR)

REV: A RELEASED: 09−NOV−2001
CHANGE DESIGNATOR: INITIAL RELEASE

Minor changes from Roger
APPROVAL HISTORY

07−NOV−2001	ELAINE SNARY (BRISTOL, UK)
09−NOV−2001	PATRICK BLOUET (TPA DESIGN DMD)
08−NOV−2001	FRED HOMEWOOD (STAR ADV ARCHITECTURE BR)
09−NOV−2001	ANTHONY JARVIS (USA DESIGN CENTERS)
07−NOV−2001	NICHOLAS PAVEY (R&D BRISTOL)
08−NOV−2001	ROGER SHEPHERD (STAR ADV ARCHITECTURE BR)
07−NOV−2001	LIZ SHEPPERD (QMS, BR)
07−NOV−2001	TREFOR SOUTHWELL (GPD DESIGN, BR)
09−NOV−2001	ALEX STARR (CMG) (USA DESIGN GROUP)

No References

CUSTOM ATTRIBUTES

St_title	ST220 INSTRUCTION SET ARCHITECTURE
Change designator	INITIAL RELEASE
Status	ACTIVE
Cycle type	ACTIVE
Dispatcher	SNARYE
Working Vault	BRI
Replication sites	
Classification	COMPANY CONFIDENTIAL
Technology	HCMOS9
Group Division	DVD
Mask set design rev	
Originator	TREFOR SOUTHWELL

PRELIMINARY DATA

ST200 VLIW Series

ST220 Instruction Set Architecture

Lx Technology Categorization: Standard Project

Last updated 6 November

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

Publication number ADCS 7345681
Issued by the MCDT Documentation Group on behalf of the STMicroelectronics Group.

STMicroelectronics GROUP OF COMPANIES
Australia - Brazil - Finland - China - France - Germany - Hong Kong - India - Italy - Japan - Malaysia
Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - U.S.A.

http://www.st.com

Contents

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:

ST200 Tools User Manual

This manual describes the software provided as part of the ST200 tools. It supports the development of ST200 applications for embedded systems. Applications may be developed in either a stand-alone environment, or under the OS200 Real-Time Operating System.

This manual also contains reference material relating to the ST200 MicroToolset.

ST200 Programming Manual

This manual describes the Programming guide.

ST200 Cross Development Manual

This manual describes the Cross development tools and platforms.

ST200 Command Language Reference Manual

This manual describes the Command Language.

OS200 Reference Manual

This manual is a reference guide to the OS200 Utilities, System Calls, Library and Features.

ST200 CPU Core Architecture

This manual describes the architecture of the ST200 core as used by STMicroelectronics.

ST200 Instruction Set Architecture

This manual describes the instruction set of the ST200 core as used by STMicroelectronics.

ST200 System Architecture

This manual describes the ST200 family system architecture. It is split into three volumes:

ST200 System Architecture, Volume 1: System

ST200 System Architecture, Volume 2: Bus Interfaces

ST200 System Architecture, Volume 3: I/O Devices

Device specific information is contained in the Datasheet for that device.

1 Instruction specification

1.1 Introduction

As described in detail in the *'ST220 Core Architecture Manual'*, the ST220 executes Very Long Instruction Word (VLIW) instructions known as bundles. Each bundle performs one or more operations. Operations can be thought of as simple RISC instructions.

The encoding of bundles is defined in *Section 6.2*.

The execution of bundles is described in *Section 2.2*, including the behavior of the machine when exceptions or interrupts are encountered. *Chapter 6 on page 45* describes the details of each operation, including the semantics.

The behavior of operations is specified using the notational language defined in *Section 3.1* through *Section 3.4*. The descriptions clearly identify where architectural state is updated and the latency of the operands.

A simple model of memory and control registers defined *Section 4.2* and *Section 4.3* is used when specifying the load and store operations.

2 Execution model

2.1 Introduction

This chapter is used to define 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 execeptions.

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.

Converesly 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.

2.2 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 *'ST220 Core Architecture Manual'*.

Note also that this simple model does not take into account the latency constraints of operands, and is therefore only valid for hazard free code *<Architecture Manual>*. All code generated by the compiler is hazard free.

The execution flow shown in *Figure 1* uses notation defined in *Chapter 3 on page 15*. Additional functions that have been used to abstract out details are described in *Section 2.3*.

Note that branching is achieved by changing the value of **NEXT_PC**.

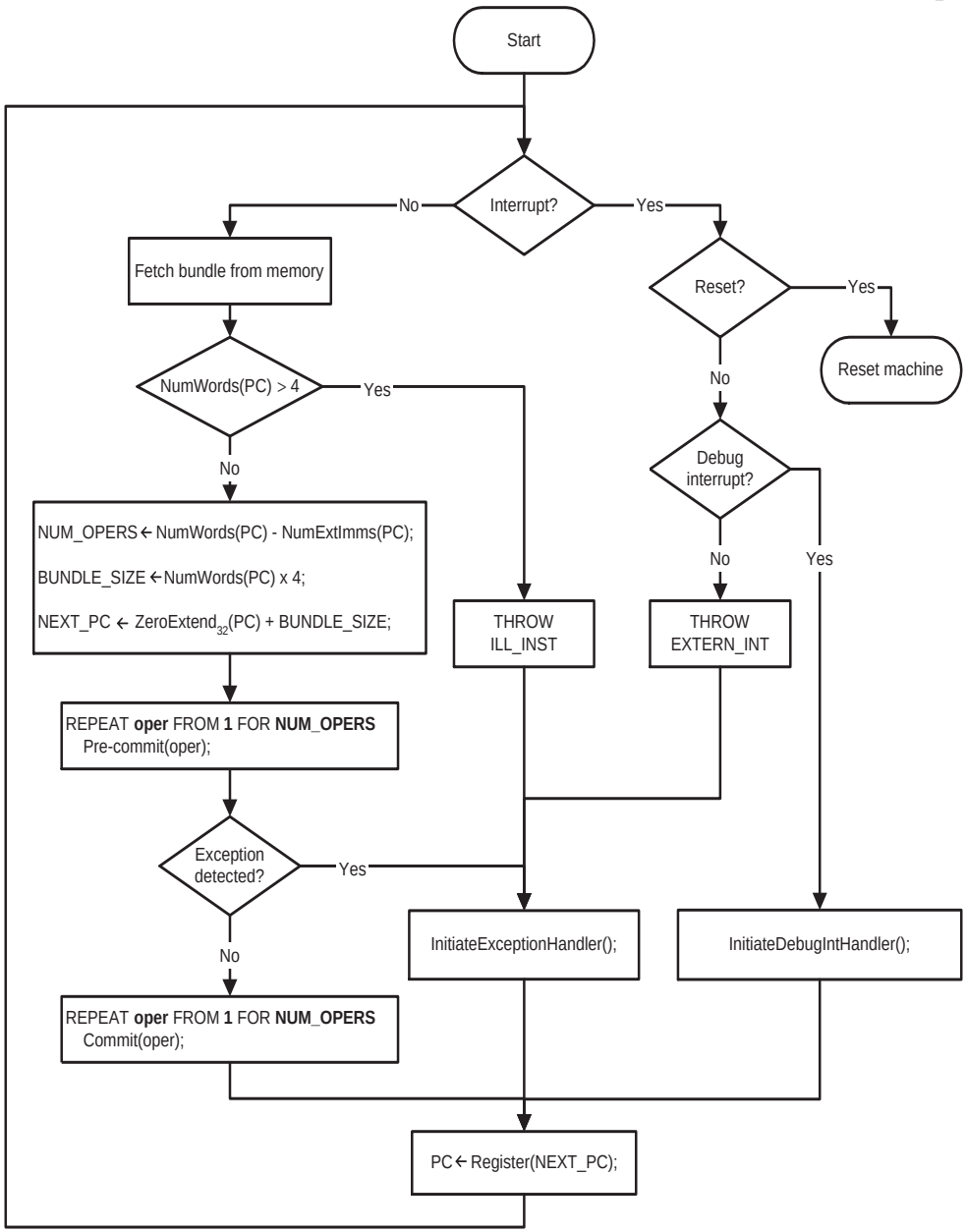


Figure 1: Execution Model

2.3 Functions

The flow chart, *Figure 1*, 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.

2.3.1 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 1: Bundle decode functions

2.3.2 Operation execution

Function	Description
Pre-commit(n)	For the operation **n**th operation in the bundle, execute the Pre-commit phase (*Section 6.3*)[A].
Commit(n)	For the operation **n**th operation in the bundle, execute the Commit phase (*Section 6.3*)[A].

Table 2: Operation execution functions

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

2.3.3 Exceptional cases

Function	Description
InitiateExceptionHandler()	Execute the statements defined in *Section 5.2*.
InitiateDebugIntHandler()	Execute the statements defined in *Section 5.3*.

Table 3: Operation execution functions

3 Specification notation

3.1 Overview

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

- A simple variable and type system (see *Section 3.2*)
- Expressions (see *Section 3.3*)
- Statements (see *Section 3.4*)
- Notation for the architectural state of the machine (see *Section 3.5*)

Additional mechanisms are defined to model memory (*Section 4.2*), control registers (*Section 4.3*), and cache instructions (*Section 4.4*).

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.

3.2 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.

3.2.1 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.

3.2.2 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**.

3.2.3 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 2^i to the integral value of the bit-field. The integral value of the bit-field as a whole is an integer in the range $[0, 2^b)$.

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

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.

3.2.4 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 $i^{th.}$ 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.

3.3 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.

3.3.1 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 4*.

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 4: Standard dyadic operators

The standard monadic operators are described in *Table 5*.

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

Table 5: 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**:

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

3.3.2 Integer shift operators

The available integer shift operators are listed in *Table 6*.

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

Table 6: Shift operators

The shift operators are defined on integers as follows where $\mathbf{b} \geq \mathbf{0}$:

$$n \ll b = n \times 2^{b}$$

$$n \gg b = \begin{cases} n / 2^{b} & \text{where} \quad n \geq 0 \\ (n - 2^{b} + 1) / 2^{b} & \text{where} \quad n < 0 \end{cases}$$

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**.

3.3.3 Integer bitwise operators

The available integer bitwise operators are listed in *Table 7*.

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

Table 7: 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** (∧), **OR** (∨), **XOR** (⊕) and **NOT** (~) 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 **b** ≥ **0** and **m** > **0**:

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

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

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

3.3.4 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 8: Relational operators

3.3.5 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 9: Boolean operators

3.3.6 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 3.4.2: Assignment on page 24*.

Functions can contain side-effects.

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 $\mathbf{[-2^{n-1}, 2^{n-1})}$ while the unsigned number range is $\mathbf{[0, 2^{n})}$.

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 10: Integer conversion operators

These two functions are defined as follows, where $\mathbf{n > 0}$:

$$\text{ZeroExtend}_n(i) = i_{\langle 0 \text{ FOR } n \rangle}$$

$$\text{SignExtend}_n(i) = \begin{cases} i_{\langle 0 \text{ FOR } n \rangle} & \text{where} \quad i_{\langle n-1 \rangle} = 0 \\ i_{\langle 0 \text{ FOR } (n-1) \rangle} - 2^n & \text{where} \quad i_{\langle n-1 \rangle} = 1 \end{cases}$$

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 11* 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 $\mathbf{i_{<0>}}$
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 $\mathbf{i_{<0\ FOR\ 32>}}$

Table 11: Conversion operators from integers to bit-fields

3.4 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.

3.4.1 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.

3.4.2 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 3.3: Expressions on page 17*. 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 12: Scalar architectural state on page 28*). 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);
```

3.4.3 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
```

3.4.4 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**.

3.4.5 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 *ST220 Core Architecture Manual*.

3.4.6 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);
```

3.5 Architectural state

The *Architectural state* chapter of the *ST220 Core Architecture* manual contains a full description of the visible state. The notations used in the specification to refer to this state are summarized in *Table 12* and *Table 13*. 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 $\mathbf{R_0}$ reads as zero Assignments to $\mathbf{R_0}$ are ignored
LR	32 bits	Link Register, synonym for $\mathbf{R_{63}}$
B_i where **i** is in **[0, 7]**	1 bit	8 x 1-bit Branch Registers

Table 12: 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 *Control Registers* chapter of the *ST220 Core Architecture* manual
MEM[i] where **i** is in **[0, 2^{32})**	8 bits	2^{32} bytes of memory

Table 13: Array architectural state

4 Memory and control registers

4.1 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 (*Memory access protection units* chapter of the *ST220 Core Architecture* manual), 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 (*Memory access protection units* chapter of the *ST220 Core Architecture* manual), 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**

Table 14: Support functions

Function	Description
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**

Table 14: Support functions

4.2 Memory Model

The instruction specification uses a simple model of memory. It assumes, for example, that any caches are not architecturally visibile. 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.

4.2.1 Support functions

The specification of the memory instructions relies on the support functions listed in *Table 14 on page 31*. These functions are used to model the behavior of the Data Protection Unit and Instruction Protection Unit described in *Memory access protection units* chapter of the *ST220 Core Architecture* manual.

4.2.2 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)	Returns **n**-bits from **address** (can generate BusError exception)
DisReadCheckMemory$_n$(address)	Throws any non-BusError exception generated by an **n**-bit dismissable read from **address**
DisReadMemory$_n$(address)	Returns either **n**-bits from **address** or **0** (can generate BusError exception)

Table 15: 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.

```
ReadCheckMemory_n(a);
```

is equivalent to:

```
IF (PSW[DPU_ENABLE] AND
    PSW[DBREAK_ENABLE] AND
    DataBreakPoint(a)) THROW DBREAK, a;
IF (Misaligned_n(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 dismissable read:

```
DisReadCheckMemory_n(a);
```

is equivalent to:

```
IF (PSW[DPU_ENABLE] AND
    PSW[DBREAK_ENABLE] AND
    DataBreakPoint(a)) THROW DBREAK, a;
IF (Misaligned_n(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$_n$** function 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 an integer result returns the read bit-field value. If the read memory value is to be interpreted as signed, then a sign-extension should be used on the result. The assignment:

```
result ← ReadMemory_n(a);
```

is equivalent to:

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

The **DisReadMemory$_n$** performs the same functionality for a dismissable read from memory. The assignment:

```
result ← DisReadMemory_n(a);
```

is equivalent to:

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

4.2.3 Prefetching memory

The following procedure is provided to denote memory prefetch.

Function	Description
PrefetchMemory(address)	Prefetch memory if possible.

Table 16: 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 (NOT Misaligned_n(a)) {
  IF (PSW[DPU_ENABLE]) {
    IF (NOT DPUNoTranslation(a) AND
        NOT ReadAccessViolation(a))
      Prefetch(a);
  }
  ELSE
    Prefetch(a);
}
```

where **Prefetch** is a cache operation defined in *Section 4.4: Cache model on page 39*. Prefetching memory will not generate any exceptions.

4.2.4 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 17: 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.

```
WriteCheckMemory_n(a);
```

is equivalent to:

```
IF (PSW[DPU_ENABLE] AND
    PSW[DBREAK_ENABLE] AND
    DataBreakPoint(a)) THROW DBREAK, a;
IF (Misaligned_n(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:

```
WriteMemory_n(a, value);
```

is equivalent to:

```
width ← n / 8;
```

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

4.3 Control register model

4.3.1 Reading 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
ReadCheckCReg(address)	Throws any exception generated by reading from **address** in the control register space
ReadCReg(address)	Reads from the control register mapped to **address**
DisReadCheckCReg(address)	Detect any data breakpoints for the dismissable read to **address** in the control register space

Table 18: 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 (PSW[DPU_ENABLE] AND
    PSW[DBREAK_ENABLE] AND
    DataBreakPoint(a)) THROW DBREAK, a;
IF (UndefinedCReg(a)) THROW CREG_NO_MAPPING, a;
index ← CRegIndex(a);
IF (CRegReadAccessViolation(index))
  THROW CREG_ACCESS_VIOLATION, a;
```

If the access to control register space is dismissable then data breakpoints need to be checked:

```
DisReadCheckCReg(a);
```

is equivalent to:

```
IF (PSW[DPU_ENABLE] AND
    PSW[DBREAK_ENABLE] AND
    DataBreakPoint(a)) THROW DBREAK, a;
```

The control register file is denoted **CR**. The function **ReadCReg** is provided:

```
result ← ReadCReg(a);
```

is equivalent to:

```
index ← CRegIndex(a);
result ← CR[index];
```

4.3.2 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 19: 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 (PSW[DPU_ENABLE] AND
    PSW[DBREAK_ENABLE] AND
    DataBreakPoint(a)) THROW DBREAK, a;
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;
```

4.4 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 *Memory Subsystem* chapter of the *ST220 Core Architecture* manual.

Procedure	Description
PurgeIns( )	Invalidate the entire instruction cache
Sync( )	Data memory subsystem synchronisation 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

Table 20: Procedures to model cache operations

5 Traps: exceptions and interrupts

5.1 Introduction

The flow diagram, *Figure 1* in *Section 2.2*, 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.

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 is thrown in a bundle. However, only the highest priority exception (as defined in *Traps: exceptions and interrupts* chapter of the *ST220 Core Architecture* manual) is passed to the handler.

Therefore, taking an exception can be summarized as:

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

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

SAVED_PSW ← PSW;        // Save the PSW and PC
SAVED_PC ← 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. 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** when the exception was generated.

5.3 Debug interrupt handling

A debug interrupt is handled differently to other external interrupts. A full description of debug interrupts and the handler can be found in *Debugging support* chapter of the *ST220 Core Architecture* manual.

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 ← 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
```

6 Instruction set

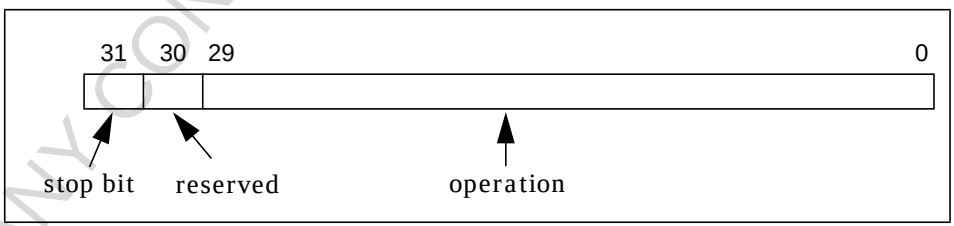
6.1 Introduction

This chapter contains descriptions of all the operations and macros (pseudo-operations) in the ST220 instruction set. *Section 6.2* has been included in order to describe how operations are encoded in the context of bundles. The architectural reasons for the encoding can be found in the '*ST220 Core Architecture Manual*'.

6.2 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:



6.2.1 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

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 21: 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:

$$(\mathrm{ZeroExtend}_{23}(\mathrm{extension}) \ll 9) + \mathrm{ZeroExtend}_{9}(i);$$

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

Otherwise **Imm** returns:

$$\mathrm{SignExtend}_{9}(i);$$

6.2.2 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.

6.3 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 (*ST220 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 3.4*) that define the operation. The notation used is defined in *Chapter 3 on page 15*. The table is divided into two parts by the commit point:

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 6.2.2*.
- 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.

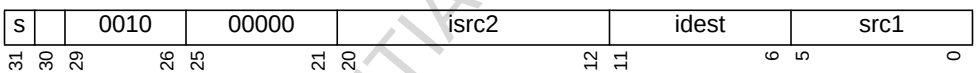
6.4 Example operations

6.4.1 add Immediate

The specification for this operation is shown below:

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.

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 6.2*).
  - 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 execption occurs in the bundle.

The statements themselves are organized into 3 stages as follows:

1 The first 2 statements read the required source information:

$$\mathbf{operand1} \leftarrow \mathbf{SignExtend_{32}(R_{src1});}$$
$$\mathbf{operand2} \leftarrow \mathbf{SignExtend_{32}(Imm(isrc2));}$$

The first statement reads the value of the $\mathbf{R_{src1}}$ 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 6.2.1*). 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:

$$\mathbf{result} \leftarrow \mathbf{operand1 + operand2;}$$

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 3.3.6*) 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.

6.5 Macros

The following are the currently implemented pseudo-operations.

`mov, mtb, mfb, nop, return, syncins, zxtb`

They are defined as specific instances of existing operations as shown below (note that **mov** appears in both the **Register** and **Immediate** formats):

	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
nop	s		0000				00000								000000						000000						000000					
mov	s		0000				00000								dest						src2						000000					
mtb	s		00011					1100				bdest									000000						src1					
mov	s		0010				00000					isrc2									idest						000000					
zxtb	s		0010				01001					011111111									idest						src1					
mfb	s		011			001			scond						000001						idest						000000					
return	s		110			0011																										
syncins	s		110			0010						00000000000000000000																				

Figure 2: Macros

Syntatically they are equivalent to:

```
mov:    add Rdest = R0, Rsrc2 / add Rdest = R0, isrc2
mtb:    orl Bbdest = Rsrc1, R0
mfb:    slctf Ridest = Bscond, R0, 1
nop:    add R0 = R0, R0
zxtb:   and Ridest = Rsrc1, 0xFF
```

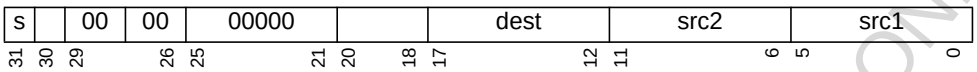
```
return:  goto LR
syncins: goto 0
```

6.6 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 3 on page 15*.

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.

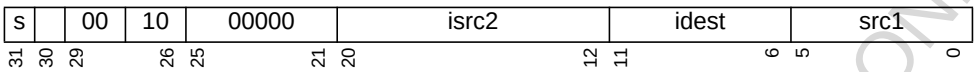
No latency constraints.

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.

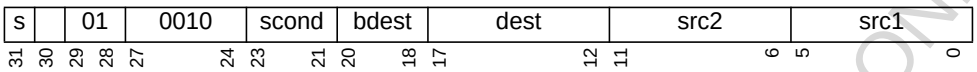
No latency constraints.

Exceptions:

None.

addcg

addcg R_{dest}, B_{bdest} = R_{src1}, R_{src2}, B_{scond}



Semantics:

operand1 ← SignExtend$_{32}$(R_{src1});
operand2 ← SignExtend$_{32}$(R_{src2});
carryin ← ZeroExtend$_{1}$(B_{scond});
result ← (operand1 + operand2) + carryin;
carryout ← result$_{< 32 >}$;

R_{dest} ← Register(result);
B_{bdest} ← Bit(carryout);

Description:

Add with carry and generate carry

Restrictions:

No address/bundle restrictions.

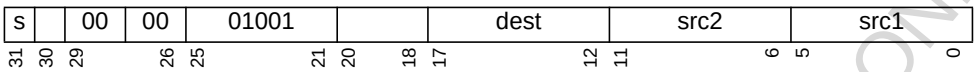
No latency constraints.

Exceptions:

None.

and Register

and $R_{dest} = R_{src1}, R_{src2}$



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.

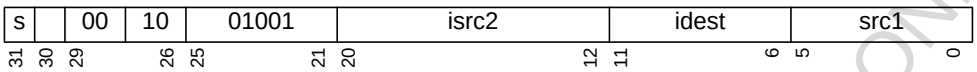
No latency constraints.

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.

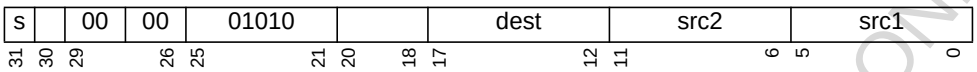
No latency constraints.

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.

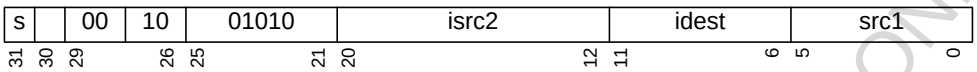
No latency constraints.

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.

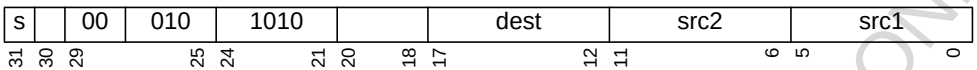
No latency constraints.

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.

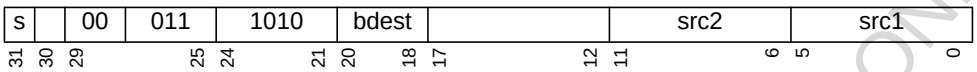
No latency constraints.

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.

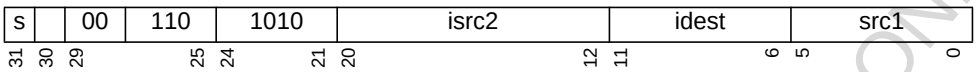
No latency constraints.

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.

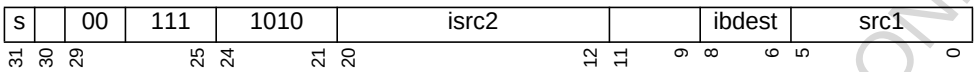
No latency constraints.

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.

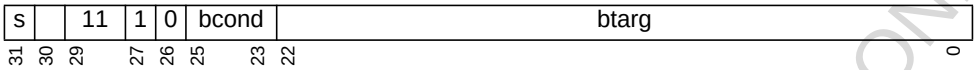
No latency constraints.

Exceptions:

None.

br

br B_{bcond}, btarg



Semantics:

operand1 ← $ZeroExtend_1(B_{bcond})$;
operand2 ← $SignExtend_{23}$(btarg);

IF (operand1 ≠ 0)
NEXT_PC ← ($ZeroExtend_{32}$(PC) + BUNDLE_SIZE) + (operand2 << 2);

Description:

Branch

Restrictions:

Must be the first syllable of a bundle.

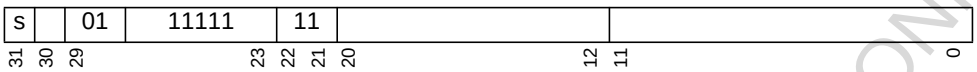
Instructions writing B_{bcond} must be followed by 2 bundles before this instruction can be issued.

Exceptions:

None.

break

break



Semantics:

```
THROW ILL_INST;
```

Description:

Break

Restrictions:

No address/bundle restrictions.

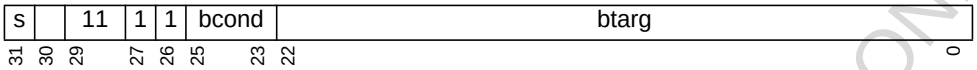
No latency constraints.

Exceptions:

ILL_INST

brf

brf B_{bcond}, btarg



Semantics:

operand1 ← $ZeroExtend_1(B_{bcond})$;
operand2 ← $SignExtend_{23}$(btarg);

IF (operand1 = 0)
NEXT_PC ← ($ZeroExtend_{32}$(PC) + BUNDLE_SIZE) + (operand2 << 2);

Description:

Branch false

Restrictions:

Must be the first syllable of a bundle.

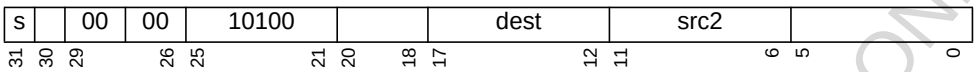
Instructions writing B_{bcond} must be followed by 2 bundles before this instruction can be issued.

Exceptions:

None.

bswap Register

bswap R_{dest} = R_{src2}



Semantics:

operand1 ← ZeroExtend$_{32}$(R_{src2});
byte0 ← operand1$_{< 0 FOR 8 >}$;
byte1 ← operand1$_{< 8 FOR 8 >}$;
byte2 ← operand1$_{< 16 FOR 8 >}$;
byte3 ← operand1$_{< 24 FOR 8 >}$;
result ← ((byte0 << 24) ∨ (byte1 << 16)) ∨ ((byte2 << 8) ∨ byte3);

R_{dest} ← Register(result);

Description:

Byte swap

Restrictions:

No address/bundle restrictions.

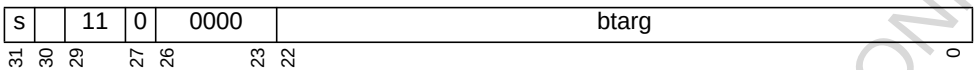
No latency constraints.

Exceptions:

None.

call Immediate

call LR = btarg



Semantics:

operand1 ← SignExtend$_{23}$(btarg);
NEXT_PC ← (ZeroExtend$_{32}$(PC) + BUNDLE_SIZE) + (operand1 << 2); LR ← Register(ZeroExtend$_{32}$(PC) + BUNDLE_SIZE);

Description:

Jump and link

Restrictions:

Must be the first syllable of a bundle.

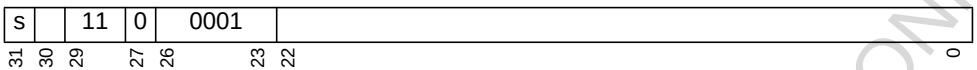
No latency constraints.

Exceptions:

None.

call Link Register

call LR = LR



Semantics:

NEXT_PC ← $ZeroExtend_{32}$(LR); LR ← Register($ZeroExtend_{32}$(PC) + BUNDLE_SIZE);

Description:

Jump (using Link Register) and link

Restrictions:

Must be the first syllable of a bundle.

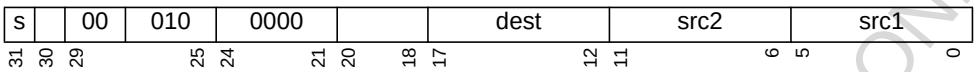
Instructions writing LR must be followed by 3 bundles before this instruction can be issued.

Exceptions:

None.

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.

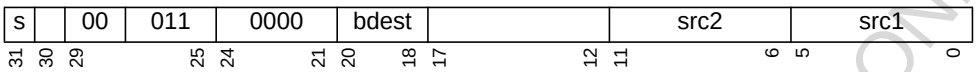
No latency constraints.

Exceptions:

None.

cmpeq Branch Register - Register

cmpeq B_{bdest} = R_{src1}, R_{src2}



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.

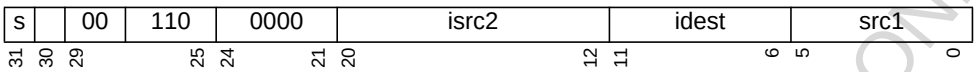
No latency constraints.

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.

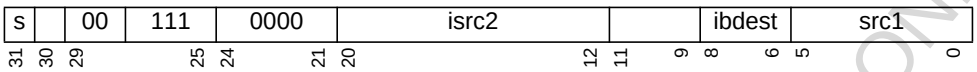
No latency constraints.

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.

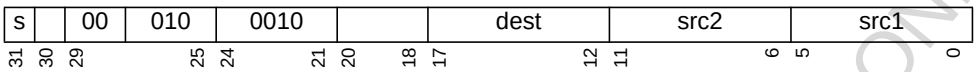
No latency constraints.

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.

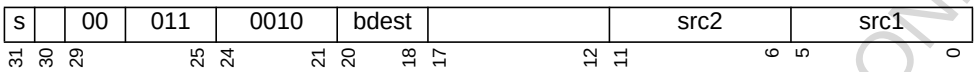
No latency constraints.

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.

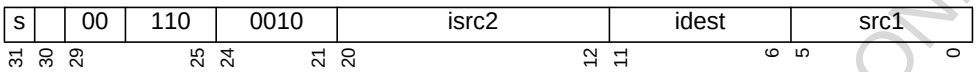
No latency constraints.

Exceptions:

None.

cmpge Register - Immediate

cmpge R_{idest} = R_{src1}, isrc2



Semantics:

operand1 ← $SignExtend_{32}(R_{src1})$;
operand2 ← $SignExtend_{32}$(Imm(isrc2));
result ← operand1 ≥ operand2;

R_{idest} ← Register(result);

Description:

Signed compare equal or greater than

Restrictions:

No address/bundle restrictions.

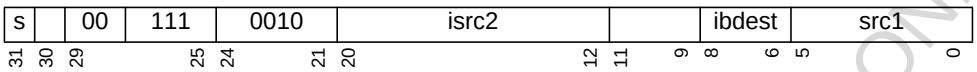
No latency constraints.

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.

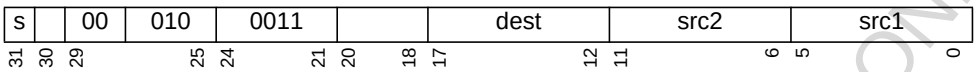
No latency constraints.

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.

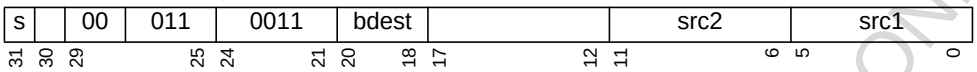
No latency constraints.

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.

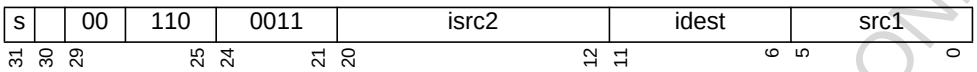
No latency constraints.

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.

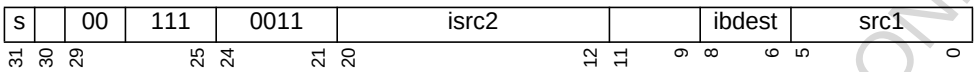
No latency constraints.

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.

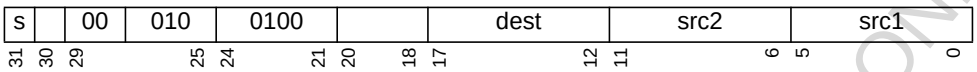
No latency constraints.

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.

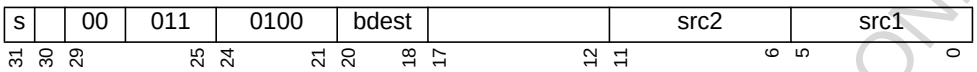
No latency constraints.

Exceptions:

None.

cmpgt Branch Register - Register

cmpgt B_{bdest} = R_{src1}, R_{src2}



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.

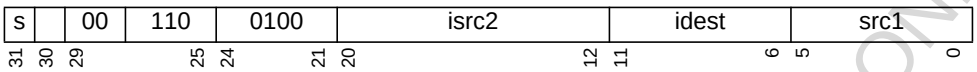
No latency constraints.

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.

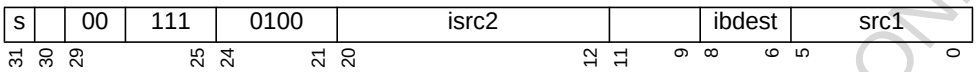
No latency constraints.

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.

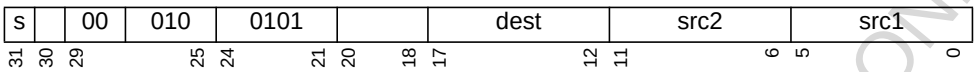
No latency constraints.

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.

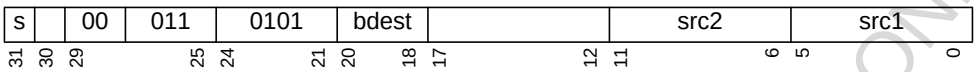
No latency constraints.

Exceptions:

None.

cmpgtu Branch Register - Register

cmpgtu B_{bdest} = R_{src1}, R_{src2}



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.

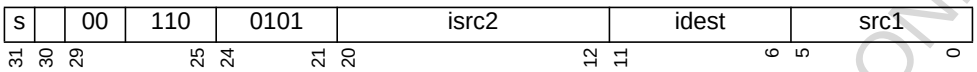
No latency constraints.

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.

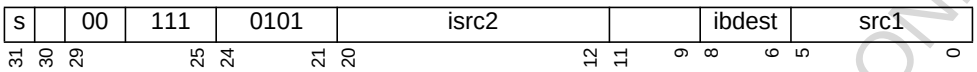
No latency constraints.

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.

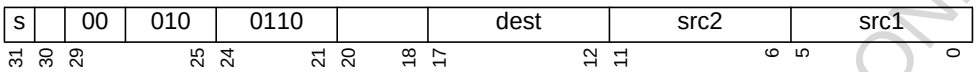
No latency constraints.

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.

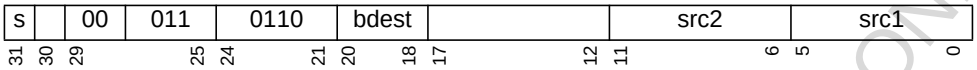
No latency constraints.

Exceptions:

None.

cmple Branch Register - Register

cmple $B_{bdest} = R_{src1}, R_{src2}$



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.

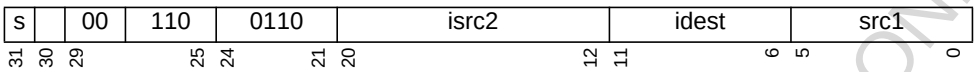
No latency constraints.

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.

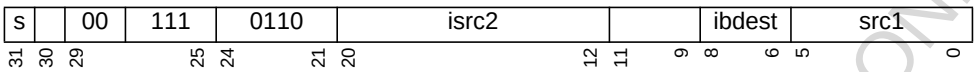
No latency constraints.

Exceptions:

None.

cmple Branch Register - Immediate

cmple $B_{ibdest} = R_{src1}$, isrc2



Semantics:

operand1 ← $SignExtend_{32}(R_{src1})$;
operand2 ← $SignExtend_{32}$(Imm(isrc2));
result ← operand1 ≤ operand2;

B_{ibdest} ← Bit(result);

Description:

Signed compare equal or less than

Restrictions:

No address/bundle restrictions.

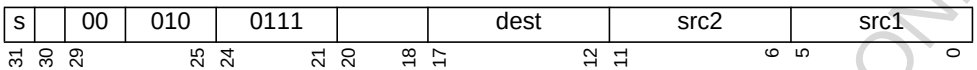
No latency constraints.

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.

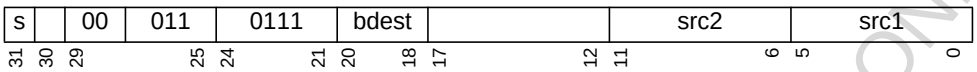
No latency constraints.

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.

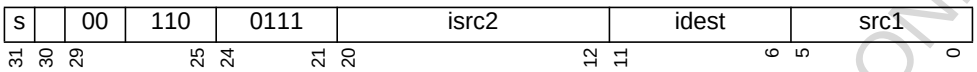
No latency constraints.

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.

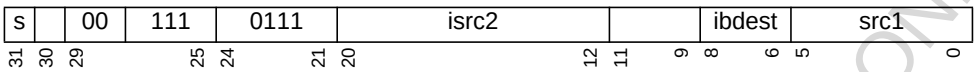
No latency constraints.

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.

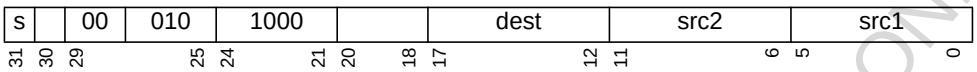
No latency constraints.

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.

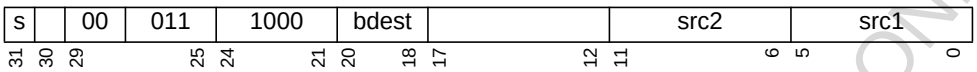
No latency constraints.

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.

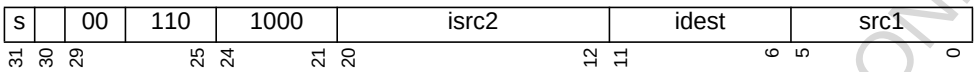
No latency constraints.

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.

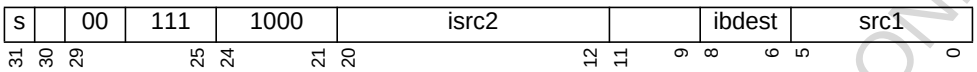
No latency constraints.

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.

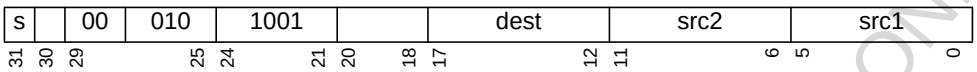
No latency constraints.

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.

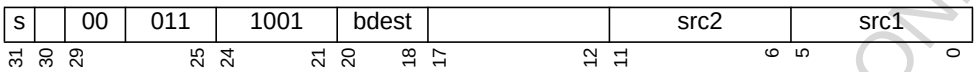
No latency constraints.

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.

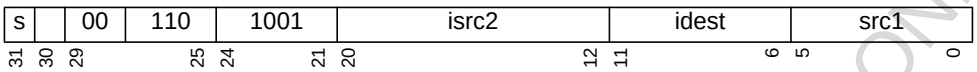
No latency constraints.

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.

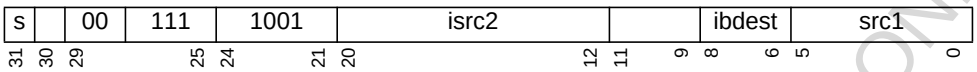
No latency constraints.

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.

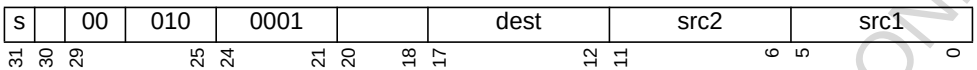
No latency constraints.

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.

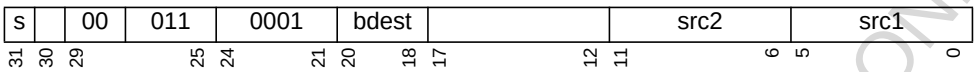
No latency constraints.

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.

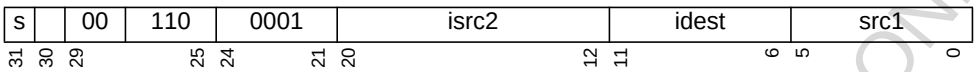
No latency constraints.

Exceptions:

None.

cmpne Register - Immediate

cmpne R_{idest} = R_{src1}, isrc2



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.

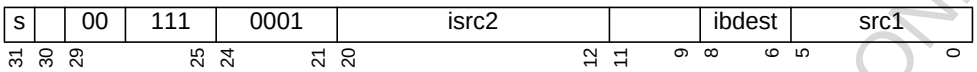
No latency constraints.

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.

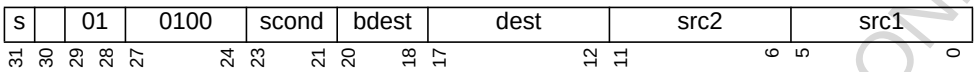
No latency constraints.

Exceptions:

None.

divs

divs R_{dest}, B_{bdest} = R_{src1}, R_{src2}, B_{scond}



Semantics:

operand1 ← $SignExtend_{32}(R_{src1})$;
operand2 ← $SignExtend_{32}(R_{src2})$;
operand3 ← $ZeroExtend_{1}(B_{scond})$;
temp ← $ZeroExtend_{32}$(operand1 × 2) + operand3;
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.

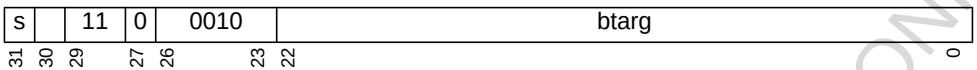
No latency constraints.

Exceptions:

None.

goto Immediate

goto btarg



Semantics:

operand1 ← SignExtend$_{23}$(btarg);
NEXT_PC ← (ZeroExtend$_{32}$(PC) + BUNDLE_SIZE) + (operand1 << 2);

Description:

Jump

Restrictions:

Must be the first syllable of a bundle.

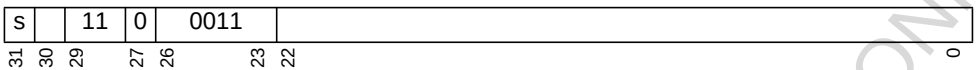
No latency constraints.

Exceptions:

None.

goto Link Register

goto LR



Semantics:

NEXT_PC ← $ZeroExtend_{32}$(LR);

Description:

Jump (using Link Register)

Restrictions:

Must be the first syllable of a bundle.

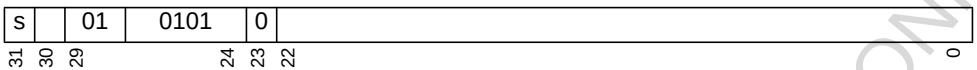
Instructions writing LR must be followed by 3 bundles before this instruction can be issued.

Exceptions:

None.

imml

imml



Semantics:

Description:

Long immediate for previous syllable

Restrictions:

Must be encoded at even word addresses.

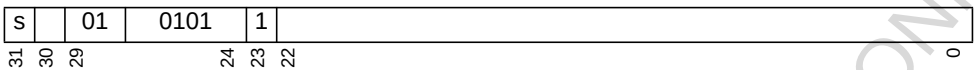
No latency constraints.

Exceptions:

None.

immr

immr



Semantics:

Description:

Long immediate for next syllable

Restrictions:

Must be encoded at even word addresses.

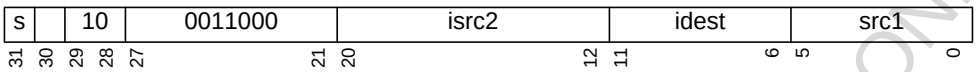
No latency constraints.

Exceptions:

None.

ldb

ldb R_{idest} = isrc2[R_{src1}]



Semantics:

```
base ← SignExtend32(Imm(isrc2));
offset ← SignExtend32(Rsrc1);
ea ← ZeroExtend32(base + offset);
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOALTION;
ELSE
        ReadCheckMemory8(ea);
```

```
result ← SignExtend8(ReadMemory8(ea));
Ridest ← Register(result);
```

Description:

Signed load byte

Restrictions:

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

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

Exceptions:

CREG_ACCESS_VIOALTION

DBREAK

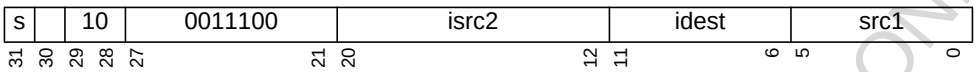
MISALIGNED_TRAP

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 (IsCRegSpace(ea))
        DisReadCheckCReg(ea);
ELSE
        DisReadCheckMemory8(ea);
```

```
IF (IsCRegSpace(ea))
        result ← 0;
ELSE
        result ← SignExtend8(DisReadMemory8(ea));
Ridest ← Register(result);
```

Description:

Signed load byte dismissable

Restrictions:

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

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

Exceptions:

DBREAK

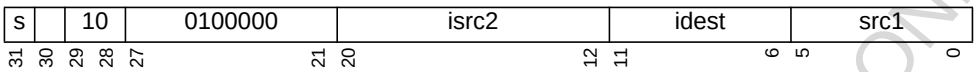
MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldbu

ldbu R_{idest} = isrc2[R_{src1}]



Semantics:

```
base ← SignExtend32(Imm(isrc2));
offset ← SignExtend32(Rsrc1);
ea ← ZeroExtend32(base + offset);
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOALTION;
ELSE
        ReadCheckMemory8(ea);
```

```
result ← ZeroExtend8(ReadMemory8(ea));
Ridest ← Register(result);
```

Description:

Unsigned load byte

Restrictions:

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

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

Exceptions:

CREG_ACCESS_VIOALTION

DBREAK

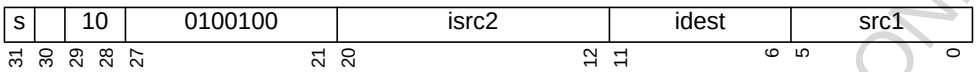
MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldbu.d

ldbu.d R_{idest} = isrc2[R_{src1}]



Semantics:

```
base ← SignExtend32(Imm(isrc2));
offset ← SignExtend32(Rsrc1);
ea ← ZeroExtend32(base + offset);
IF (IsCRegSpace(ea))
        DisReadCheckCReg(ea);
ELSE
        DisReadCheckMemory8(ea);
```

```
IF (IsCRegSpace(ea))
        result ← 0;
ELSE
        result ← ZeroExtend8(DisReadMemory8(ea));
Ridest ← Register(result);
```

Description:

Unsigned load byte dismissable

Restrictions:

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

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

Exceptions:

DBREAK

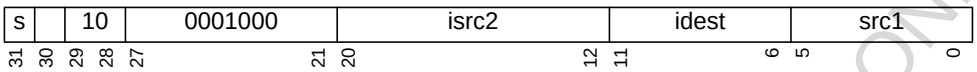
MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldh

ldh R_{idest} = isrc2[R_{src1}]



Semantics:

```
base ← SignExtend32(Imm(isrc2));
offset ← SignExtend32(Rsrc1);
ea ← ZeroExtend32(base + offset);
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOALTION;
ELSE
        ReadCheckMemory16(ea);
```

```
result ← SignExtend16(ReadMemory16(ea));
Ridest ← Register(result);
```

Description:

Signed load half-word

Restrictions:

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

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

Exceptions:

CREG_ACCESS_VIOALTION

DBREAK

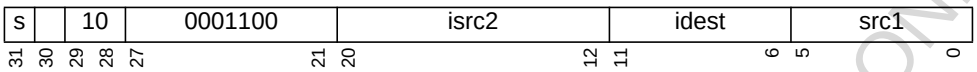
MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldh.d

ldh.d R_{idest} = isrc2[R_{src1}]



Semantics:

base ← $SignExtend_{32}$(Imm(isrc2));
offset ← $SignExtend_{32}$(R_{src1});
ea ← $ZeroExtend_{32}$(base + offset);
IF (IsCRegSpace(ea))
    DisReadCheckCReg(ea);
ELSE
    $DisReadCheckMemory_{16}$(ea);

IF (IsCRegSpace(ea))
    result ← 0;
ELSE
    result ← $SignExtend_{16}$($DisReadMemory_{16}$(ea));
R_{idest} ← Register(result);

Description:

Signed load half-word dismissable

Restrictions:

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

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

Exceptions:

DBREAK

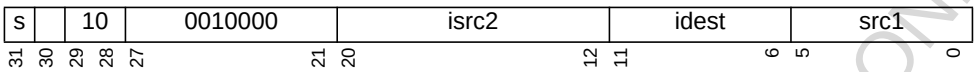
MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldhu

ldhu R_{idest} = isrc2[R_{src1}]



Semantics:

```
base ← SignExtend32(Imm(isrc2));
offset ← SignExtend32(Rsrc1);
ea ← ZeroExtend32(base + offset);
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOALTION;
ELSE
        ReadCheckMemory16(ea);
```

```
result ← ZeroExtend16(ReadMemory16(ea));
Ridest ← Register(result);
```

Description:

Unsigned load half-word

Restrictions:

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

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

Exceptions:

CREG_ACCESS_VIOALTION

DBREAK

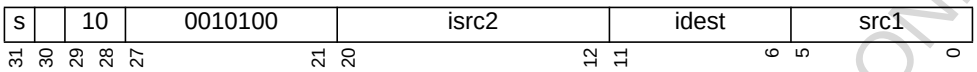
MISALIGNED_TRAP

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 (IsCRegSpace(ea))
        DisReadCheckCReg(ea);
ELSE
        DisReadCheckMemory16(ea);
```

```
IF (IsCRegSpace(ea))
        result ← 0;
ELSE
        result ← ZeroExtend16(DisReadMemory16(ea));
Ridest ← Register(result);
```

Description:

Unsigned load half-word dismissable

Restrictions:

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

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

Exceptions:

DBREAK

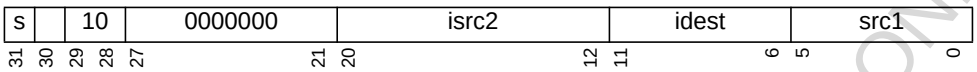
MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

ldw

ldw R_{idest} = isrc2[R_{src1}]



Semantics:

```
base ← SignExtend32(Imm(isrc2));
offset ← SignExtend32(Rsrc1);
ea ← ZeroExtend32(base + offset);
IF (IsCRegSpace(ea))
        ReadCheckCReg(ea);
ELSE
        ReadCheckMemory32(ea);
```

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

Description:

Load word

Restrictions:

Uses the load/store unit, for which only 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

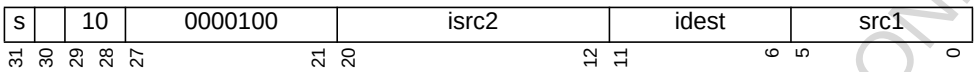
DPU_ACCESS_VIOLATION

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 (IsCRegSpace(ea))
        DisReadCheckCReg(ea);
ELSE
        DisReadCheckMemory32(ea);
```

```
IF (IsCRegSpace(ea))
        result ← 0;
ELSE
        result ← SignExtend32(DisReadMemory32(ea));
Ridest ← Register(result);
```

Description:

Load word dismissable

Restrictions:

Uses the load/store unit, for which only 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

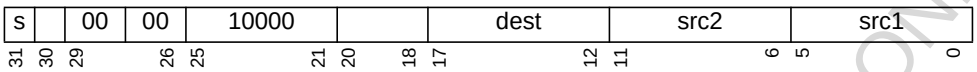
DPU_ACCESS_VIOLATION

CREG_NO_MAPPING

CREG_ACCESS_VIOLATION

max Register

max R_{dest} = R_{src1}, R_{src2}



Semantics:

operand1 ← SignExtend$_{32}$(R_{src1});
operand2 ← SignExtend$_{32}$(R_{src2});
IF (operand1 > operand2)
    result ← operand1;
ELSE
    result ← operand2;

R_{dest} ← Register(result);

Description:

Signed maximum

Restrictions:

No address/bundle restrictions.

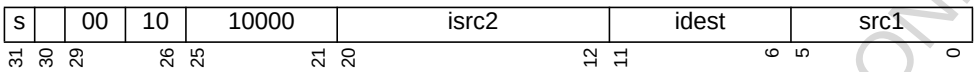
No latency constraints.

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.

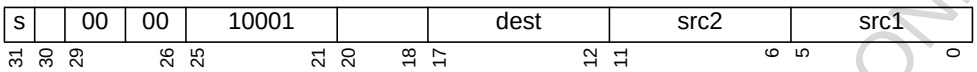
No latency constraints.

Exceptions:

None.

maxu Register

maxu R_{dest} = R_{src1}, R_{src2}



Semantics:

operand1 ← $ZeroExtend_{32}(R_{src1})$;
operand2 ← $ZeroExtend_{32}(R_{src2})$;
IF (operand1 > operand2)
    result ← operand1;
ELSE
    result ← operand2;

R_{dest} ← Register(result);

Description:

Unsigned maximum

Restrictions:

No address/bundle restrictions.

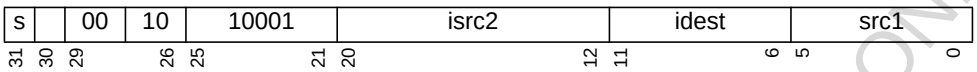
No latency constraints.

Exceptions:

None.

maxu Immediate

maxu R_{idest} = R_{src1}, isrc2



Semantics:

operand1 ← $ZeroExtend_{32}(R_{src1})$;
operand2 ← $ZeroExtend_{32}$(Imm(isrc2));
IF (operand1 > operand2)
    result ← operand1;
ELSE
    result ← operand2;

R_{idest} ← Register(result);

Description:

Unsigned maximum

Restrictions:

No address/bundle restrictions.

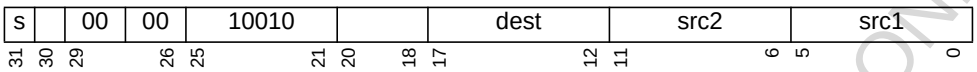
No latency constraints.

Exceptions:

None.

min Register

min R_{dest} = R_{src1}, R_{src2}



Semantics:

operand1 ← SignExtend$_{32}$(R_{src1});
operand2 ← SignExtend$_{32}$(R_{src2});
IF (operand1 < operand2)
    result ← operand1;
ELSE
    result ← operand2;

R_{dest} ← Register(result);

Description:

Signed minimum

Restrictions:

No address/bundle restrictions.

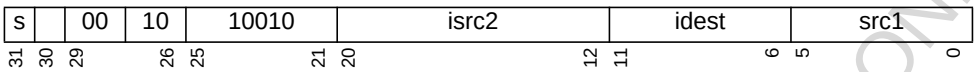
No latency constraints.

Exceptions:

None.

min Immediate

min R_{idest} = R_{src1}, isrc2



Semantics:

operand1 ← $SignExtend_{32}(R_{src1})$;
operand2 ← $SignExtend_{32}$(Imm(isrc2));
IF (operand1 < operand2)
    result ← operand1;
ELSE
    result ← operand2;

R_{idest} ← Register(result);

Description:

Signed minimum

Restrictions:

No address/bundle restrictions.

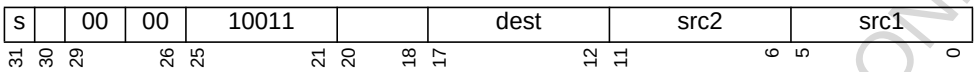
No latency constraints.

Exceptions:

None.

minu Register

minu R_{dest} = R_{src1}, R_{src2}



Semantics:

operand1 ← ZeroExtend$_{32}$(R_{src1});
operand2 ← ZeroExtend$_{32}$(R_{src2});
IF (operand1 < operand2)
    result ← operand1;
ELSE
    result ← operand2;

R_{dest} ← Register(result);

Description:

Unsigned minimum

Restrictions:

No address/bundle restrictions.

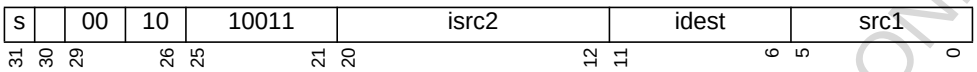
No latency constraints.

Exceptions:

None.

minu Immediate

minu $R_{idest} = R_{src1}$, isrc2



Semantics:

operand1 ← $ZeroExtend_{32}(R_{src1})$;
operand2 ← $ZeroExtend_{32}$(Imm(isrc2));
IF (operand1 < operand2)
    result ← operand1;
ELSE
    result ← operand2;

R_{idest} ← Register(result);

Description:

Unsigned minimum

Restrictions:

No address/bundle restrictions.

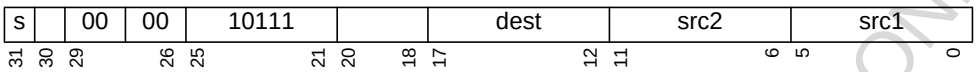
No latency constraints.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

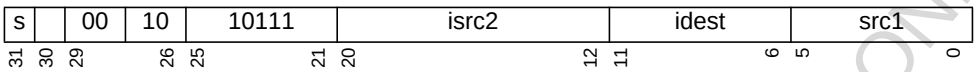
This instruction must be followed by 2 bundles before R_{dest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

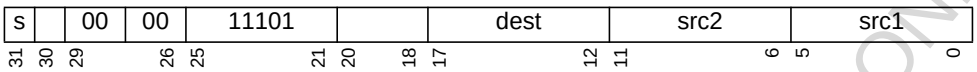
This instruction must be followed by 2 bundles before R_{idest} can be read.

Exceptions:

None.

mulhh Register

mulhh R_{dest} = R_{src1}, R_{src2}



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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

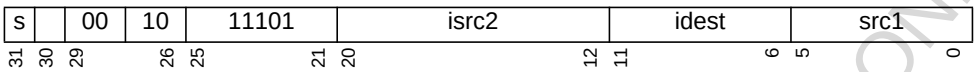
This instruction must be followed by 2 bundles before R_{dest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

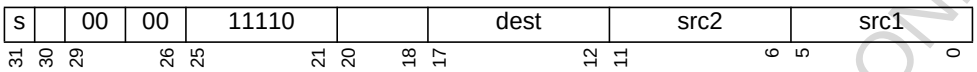
This instruction must be followed by 2 bundles before R_{idest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

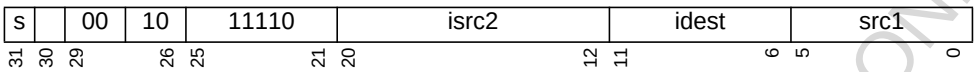
This instruction must be followed by 2 bundles before R_{dest} can be read.

Exceptions:

None.

mulhhu Immediate

mulhhu R_{idest} = R_{src1}, isrc2



Semantics:

operand1 ← $SignExtend_{32}(R_{src1})$;
operand2 ← $SignExtend_{32}(Imm(isrc2))$;
result ← $ZeroExtend_{16}(operand1 >> 16) \times ZeroExtend_{16}(operand2 >> 16)$;

R_{idest} ← Register(result);

Description:

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

Restrictions:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

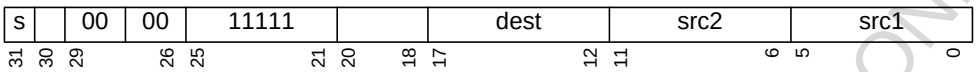
This instruction must be followed by 2 bundles before R_{idest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

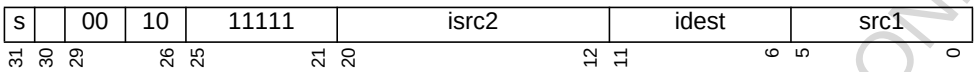
This instruction must be followed by 2 bundles before R_{dest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

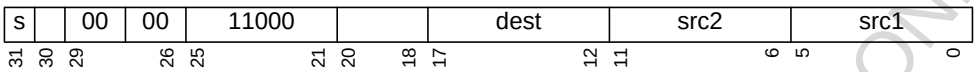
This instruction must be followed by 2 bundles before R_{idest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

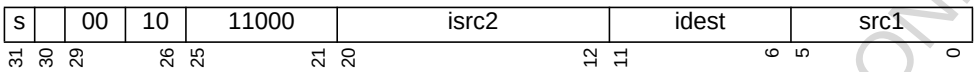
This instruction must be followed by 2 bundles before R_{dest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

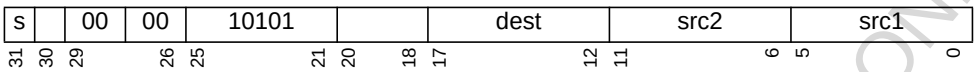
This instruction must be followed by 2 bundles before R_{idest} can be read.

Exceptions:

None.

mull Register

mull R_{dest} = R_{src1}, R_{src2}



Semantics:

operand1 ← SignExtend$_{32}$(R_{src1}); operand2 ← SignExtend$_{16}$(R_{src2}); result ← operand1 × operand2;
R_{dest} ← Register(result);

Description:

Word by half-word signed multiply

Restrictions:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

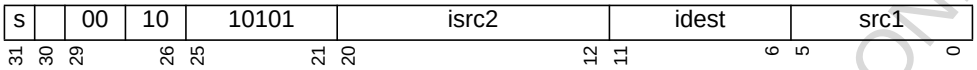
This instruction must be followed by 2 bundles before R_{dest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

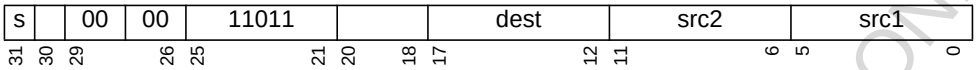
This instruction must be followed by 2 bundles before R_{idest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

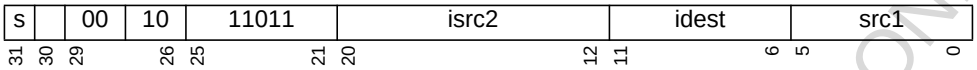
This instruction must be followed by 2 bundles before R_{dest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

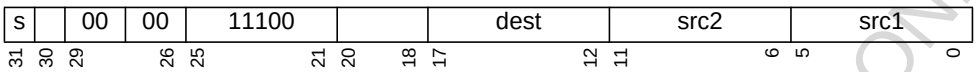
This instruction must be followed by 2 bundles before R_{idest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

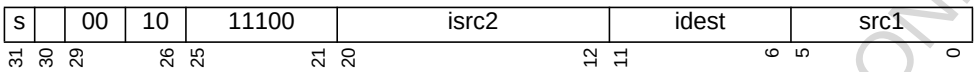
This instruction must be followed by 2 bundles before R_{dest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

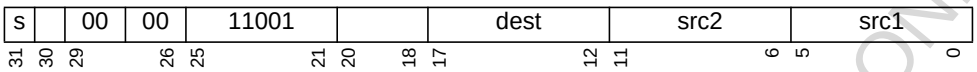
This instruction must be followed by 2 bundles before R_{idest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

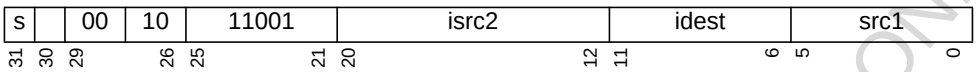
This instruction must be followed by 2 bundles before R_{dest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

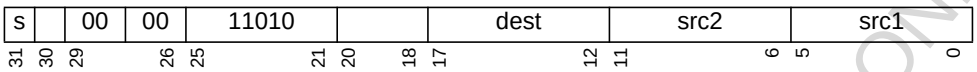
This instruction must be followed by 2 bundles before R_{idest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

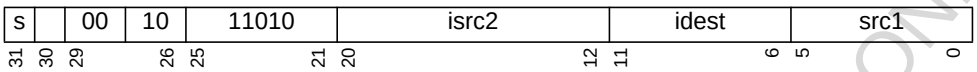
This instruction must be followed by 2 bundles before R_{dest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

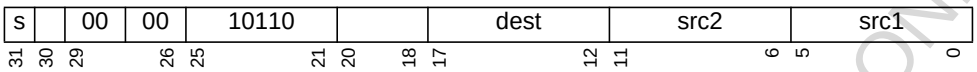
This instruction must be followed by 2 bundles before R_{idest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

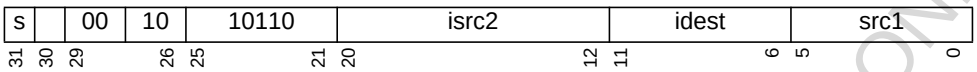
This instruction must be followed by 2 bundles before R_{dest} can be read.

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:

Must be encoded at odd word addresses.

Cannot write the link register (LR).

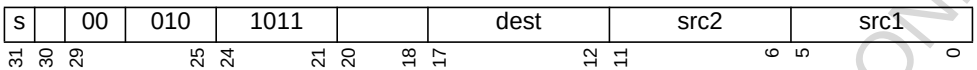
This instruction must be followed by 2 bundles before R_{idest} can be read.

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.

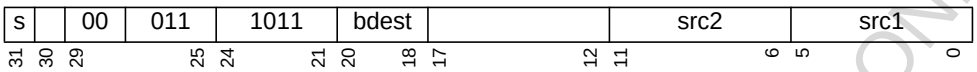
No latency constraints.

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.

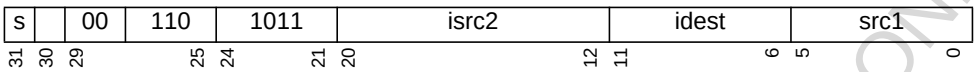
No latency constraints.

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.

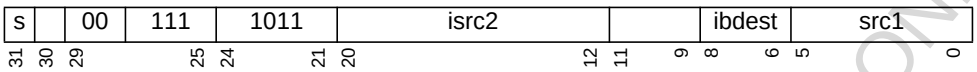
No latency constraints.

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.

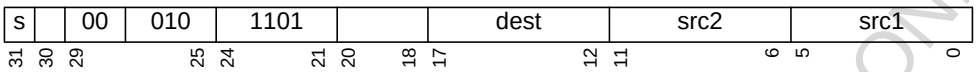
No latency constraints.

Exceptions:

None.

norl Register - Register

norl R_{dest} = R_{src1}, R_{src2}



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.

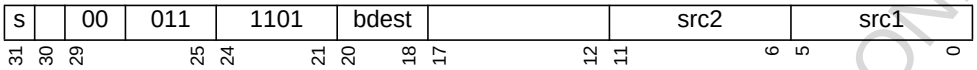
No latency constraints.

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.

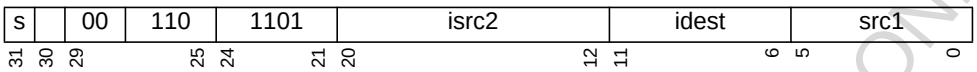
No latency constraints.

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.

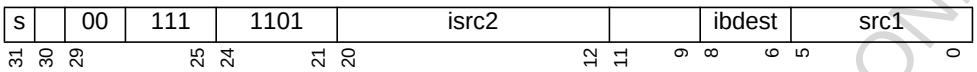
No latency constraints.

Exceptions:

None.

norl Branch Register - Immediate

norl B_{ibdest} = R_{src1}, isrc2



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.

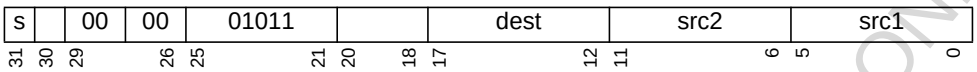
No latency constraints.

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.

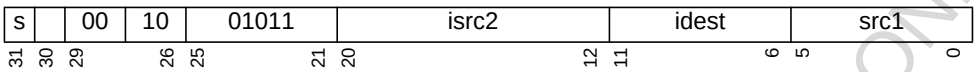
No latency constraints.

Exceptions:

None.

or Immediate

or R_{idest} = R_{src1}, isrc2



Semantics:

operand1 ← $SignExtend_{32}(R_{src1})$;
operand2 ← $SignExtend_{32}$(Imm(isrc2));
result ← operand1 ∨ operand2;

R_{idest} ← Register(result);

Description:

Bitwise or

Restrictions:

No address/bundle restrictions.

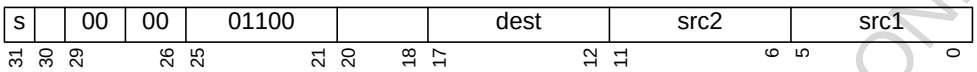
No latency constraints.

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.

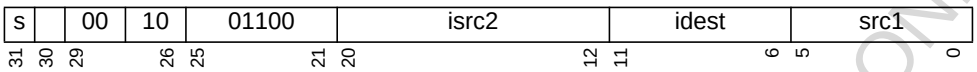
No latency constraints.

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.

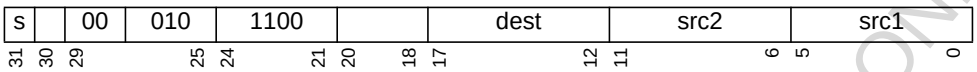
No latency constraints.

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.

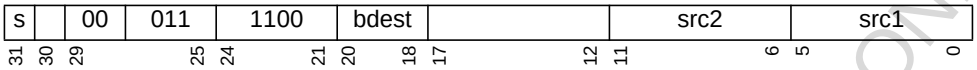
No latency constraints.

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.

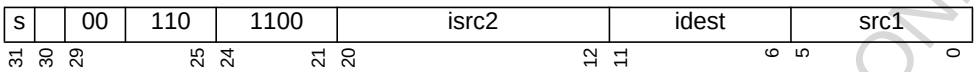
No latency constraints.

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.

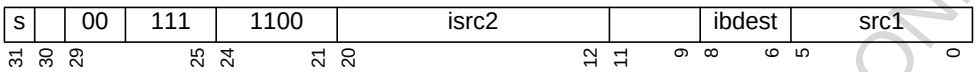
No latency constraints.

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.

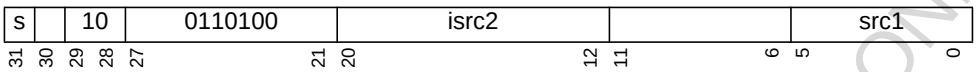
No latency constraints.

Exceptions:

None.

pft

pft isrc2[R_{src1}]



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 operation is allowed per bundle.

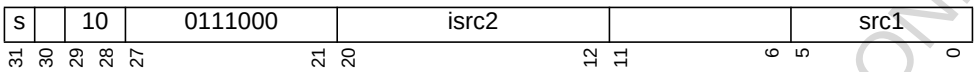
No latency constraints.

Exceptions:

None.

prgadd

prgadd isrc2[R_{src1}]



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 operation is allowed per bundle.

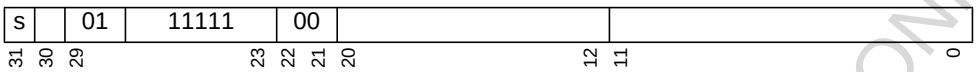
No latency constraints.

Exceptions:

None.

prgins

prgins



Semantics:

```
PurgeIns();
```

Description:

Purge the instruction cache

Restrictions:

Must be the first syllable of a bundle.

Must be followed by 3 bundles delay before issuing a syncins operation

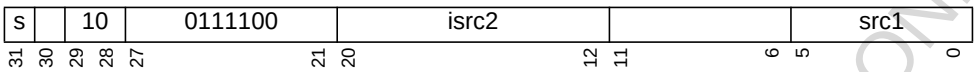
No latency constraints.

Exceptions:

None.

prgset

prgset isrc2[R_{src1}]



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 operation is allowed per bundle.

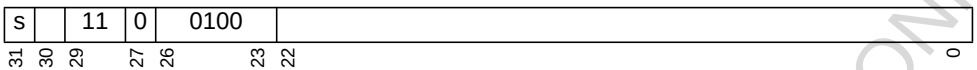
No latency constraints.

Exceptions:

None.

rfi

rfi



Semantics:

```
IF (PSW[USER_MODE])
        THROW ILL_INST;
```

```
NEXT_PC ← 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.

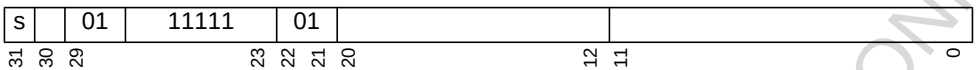
Instructions writing SAVED_PSW must be followed by 4 bundles before this instruction can be issued.

Exceptions:

ILL_INST

sbrk

sbrk



Semantics:

```
THROW SBREAK;
```

Description:

Software breakpoint

Restrictions:

No address/bundle restrictions.

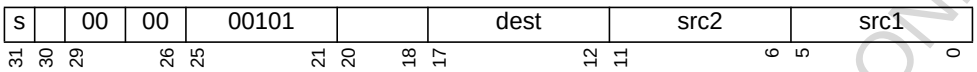
No latency constraints.

Exceptions:

SBREAK

sh1add Register

sh1add R_{dest} = R_{src1}, R_{src2}



Semantics:

operand1 ← SignExtend$_{32}$(R_{src1});
operand2 ← SignExtend$_{32}$(R_{src2});
result ← (operand1 << 1) + operand2;

R_{dest} ← Register(result);

Description:

Shift left one and accumulate

Restrictions:

No address/bundle restrictions.

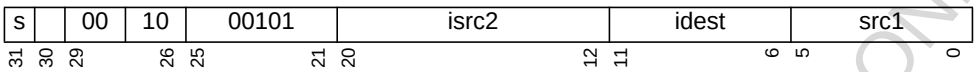
No latency constraints.

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.

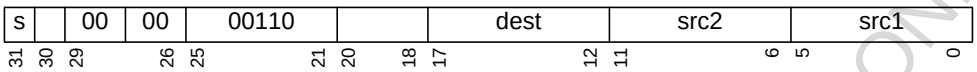
No latency constraints.

Exceptions:

None.

sh2add Register

sh2add R_{dest} = R_{src1}, R_{src2}



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.

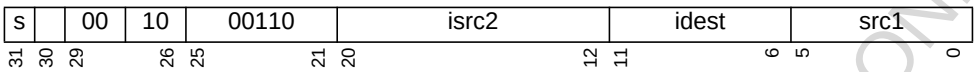
No latency constraints.

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.

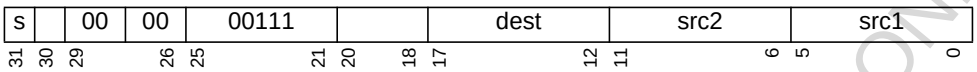
No latency constraints.

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.

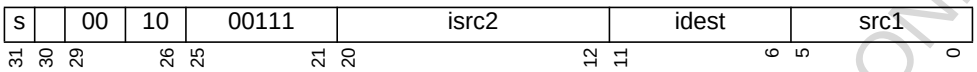
No latency constraints.

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.

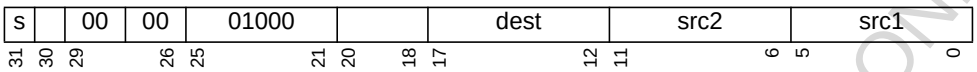
No latency constraints.

Exceptions:

None.

sh4add Register

sh4add R_{dest} = R_{src1}, R_{src2}



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.

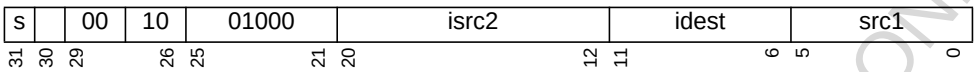
No latency constraints.

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.

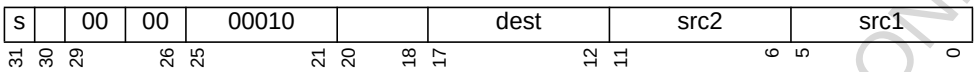
No latency constraints.

Exceptions:

None.

shl Register

shl R_{dest} = R_{src1}, R_{src2}



Semantics:

operand1 ← $SignExtend_{32}(R_{src1})$;
operand2 ← $SignExtend_{32}(R_{src2})$;
distance ← $ZeroExtend_{8}$(operand2);
result ← operand1 << distance;

R_{dest} ← Register(result);

Description:

Shift left

Restrictions:

No address/bundle restrictions.

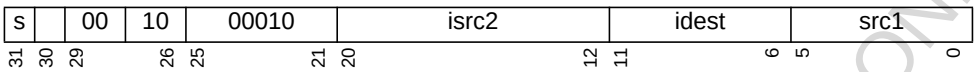
No latency constraints.

Exceptions:

None.

shl Immediate

shl R_{idest} = R_{src1}, isrc2



Semantics:

operand1 ← $SignExtend_{32}(R_{src1})$; operand2 ← $SignExtend_{32}$(Imm(isrc2)); distance ← $ZeroExtend_{8}$(operand2); result ← operand1 << distance;
R_{idest} ← Register(result);

Description:

Shift left

Restrictions:

No address/bundle restrictions.

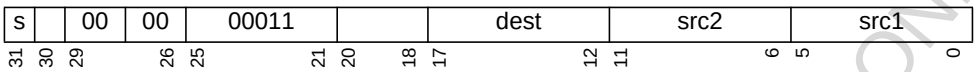
No latency constraints.

Exceptions:

None.

shr Register

shr R_{dest} = R_{src1}, R_{src2}



Semantics:

operand1 ← $SignExtend_{32}(R_{src1})$;
operand2 ← $SignExtend_{32}(R_{src2})$;
distance ← $ZeroExtend_{8}$(operand2);
result ← operand1 >> distance;

R_{dest} ← Register(result);

Description:

Arithmetic shift right

Restrictions:

No address/bundle restrictions.

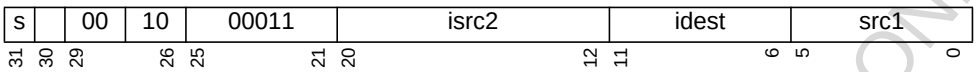
No latency constraints.

Exceptions:

None.

shr Immediate

shr R_{idest} = R_{src1}, isrc2



Semantics:

operand1 ← $SignExtend_{32}(R_{src1})$; operand2 ← $SignExtend_{32}$(Imm(isrc2)); distance ← $ZeroExtend_{8}$(operand2); result ← operand1 >> distance;
R_{idest} ← Register(result);

Description:

Arithmetic shift right

Restrictions:

No address/bundle restrictions.

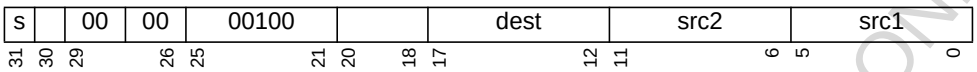
No latency constraints.

Exceptions:

None.

shru Register

shru R_{dest} = R_{src1}, R_{src2}



Semantics:

operand1 ← $ZeroExtend_{32}(R_{src1})$;
operand2 ← $SignExtend_{32}(R_{src2})$;
distance ← $ZeroExtend_{8}$(operand2);
result ← operand1 >> distance;

R_{dest} ← Register(result);

Description:

Logical shift right

Restrictions:

No address/bundle restrictions.

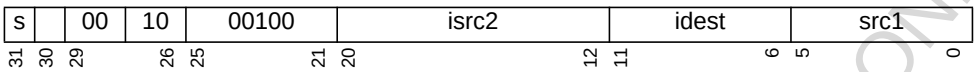
No latency constraints.

Exceptions:

None.

shru Immediate

shru R_{idest} = R_{src1}, isrc2



Semantics:

operand1 ← $ZeroExtend_{32}(R_{src1})$;
operand2 ← $SignExtend_{32}(Imm(isrc2))$;
distance ← $ZeroExtend_{8}(operand2)$;
result ← operand1 >> distance;

R_{idest} ← Register(result);

Description:

Logical shift right

Restrictions:

No address/bundle restrictions.

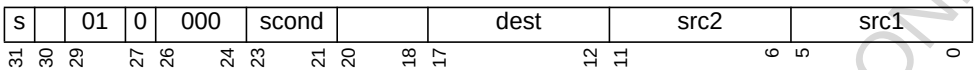
No latency constraints.

Exceptions:

None.

slct Register

slct R_{dest} = B_{scond}, R_{src1}, R_{src2}



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.

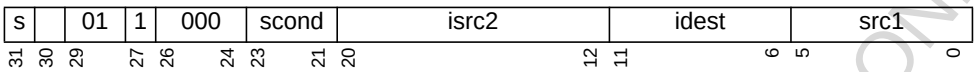
No latency constraints.

Exceptions:

None.

slct Immediate

slct R_{idest} = B_{scond}, R_{src1}, isrc2



Semantics:

operand1 ← ZeroExtend$_1$(B_{scond});
operand2 ← SignExtend$_{32}$(R_{src1});
operand3 ← SignExtend$_{32}$(Imm(isrc2));
IF (operand1 ≠ 0)
    result ← operand2;
ELSE
    result ← operand3;

R_{idest} ← Register(result);

Description:

Conditional select

Restrictions:

No address/bundle restrictions.

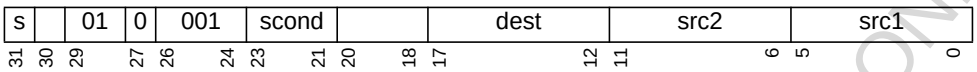
No latency constraints.

Exceptions:

None.

slctf Register

slctf R_{dest} = B_{scond}, R_{src1}, R_{src2}



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.

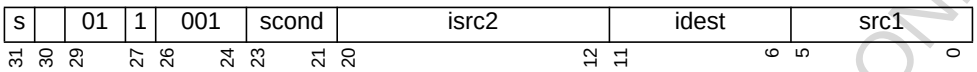
No latency constraints.

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.

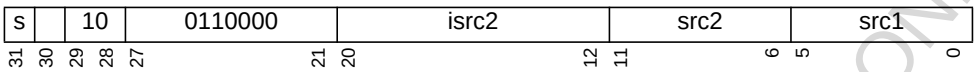
No latency constraints.

Exceptions:

None.

stb

stb isrc2[R_{src1}] = R_{src2}



Semantics:

```
base ← SignExtend32(Imm(isrc2));
offset ← SignExtend32(Rsrc1);
operand3 ← SignExtend32(Rsrc2);
ea ← ZeroExtend32(base + offset);
IF (IsCRegSpace(ea))
        THROW CREG_ACCESS_VIOALTION;
ELSE
        WriteCheckMemory8(ea);
```

```
WriteMemory8(ea, operand3);
```

Description:

Store byte

Restrictions:

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

No latency constraints.

Exceptions:

CREG_ACCESS_VIOALTION

DBREAK

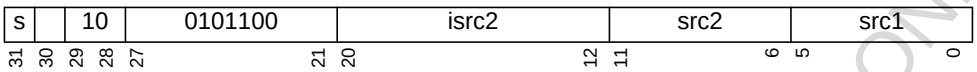
MISALIGNED_TRAP

DPU_NO_TRANSLATION

DPU_ACCESS_VIOLATION

sth

sth isrc2[R_{src1}] = R_{src2}



Semantics:

base ← $SignExtend_{32}$(Imm(isrc2));
offset ← $SignExtend_{32}$(R_{src1});
operand3 ← $SignExtend_{32}$(R_{src2});
ea ← $ZeroExtend_{32}$(base + offset);
IF (IsCRegSpace(ea))
    THROW CREG_ACCESS_VIOALTION;
ELSE
    $WriteCheckMemory_{16}$(ea);

$WriteMemory_{16}$(ea, operand3);

Description:

Store half-word

Restrictions:

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

No latency constraints.

Exceptions:

CREG_ACCESS_VIOALTION

DBREAK

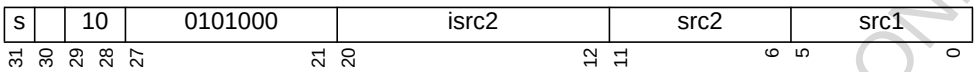
MISALIGNED_TRAP

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 (IsCRegSpace(ea))
        WriteCheckCReg(ea);
ELSE
        WriteCheckMemory32(ea);
```

```
IF (IsCRegSpace(ea))
        WriteCReg(ea, operand3);
ELSE
        WriteMemory32(ea, operand3);
```

Description:

Store word

Restrictions:

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

No latency constraints.

Exceptions:

DBREAK

MISALIGNED_TRAP

DPU_NO_TRANSLATION

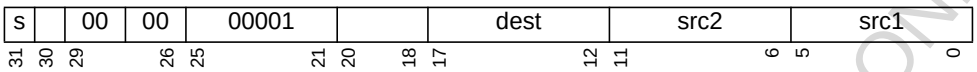
DPU_ACCESS_VIOLATION

CREG_NO_MAPPING

CREG_ACCESS_VIOLATION

sub Register

sub R_{dest} = R_{src2}, R_{src1}



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.

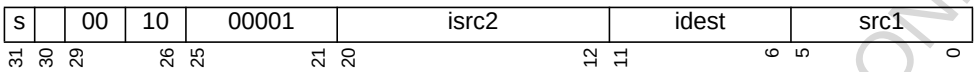
No latency constraints.

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.

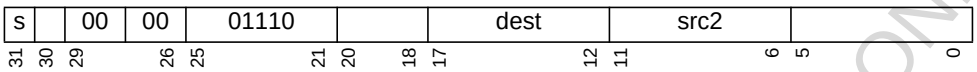
No latency constraints.

Exceptions:

None.

sxtb Register

sxtb R_{dest} = R_{src2}



Semantics:

operand1 ← $SignExtend_8(R_{src2})$; result ← operand1;
R_{dest} ← Register(result);

Description:

Sign-extend byte

Restrictions:

No address/bundle restrictions.

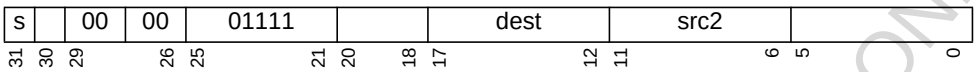
No latency constraints.

Exceptions:

None.

sxth Register

sxth R_{dest} = R_{src2}



Semantics:

operand1 ← $SignExtend_{16}(R_{src2})$; result ← operand1;
R_{dest} ← Register(result);

Description:

Sign-extend half-word

Restrictions:

No address/bundle restrictions.

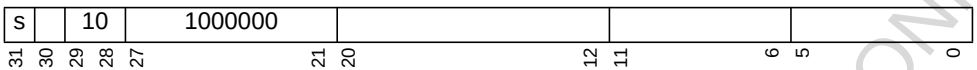
No latency constraints.

Exceptions:

None.

sync

sync



Semantics:

```
Sync();
```

Description:

Ensure synchronisation

Restrictions:

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

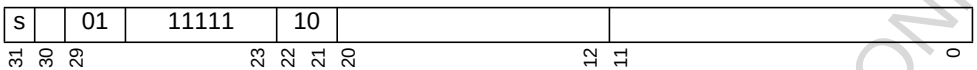
No latency constraints.

Exceptions:

None.

syscall

syscall



Semantics:

```
THROW ILL_INST;
```

Description:

System call

Restrictions:

No address/bundle restrictions.

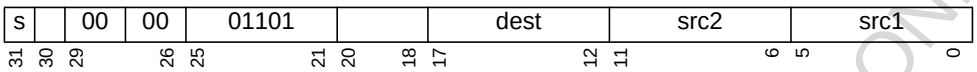
No latency constraints.

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.

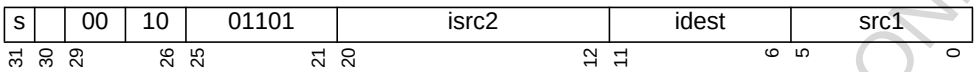
No latency constraints.

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.

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

A.3 Formats

	Bundle Stop	Cluster Bit	Format	Opcode							Immediate/ Dest									Dest/Src2						Src1					
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Int3R	s		00		00		Opcode								Dest.						Source2						Source1					
Cmp3R_Reg	s		00		010			Opcode							Dest.						Source2						Source1					
Cmp3R_Br	s		00		011			Opcode				BrDest									Source2						Source1					
Int3I	s		00		10		Opcode					Immediate									Dest.						Source1					
Cmp3I_Reg	s		00		110			Opcode				Immediate									Dest.						Source1					
Cmp3I_Br	s		00		111			Opcode				Immediate												BrDest			Source1					
IMMOP	s		01		0101				Opcode			Immediate																				
SelectR	s		01		0	Opcode			BrReg						Dest.						Source2						Source1					
SelectI	s		01		1	Opcode			BrReg			Immediate									Dest.						Source1					
ExtendedArith	s		01		Opcode				BrReg			BrDest			Dest.						Source2						Source1					
Special	s		01		11111					Opcode		Immediate																				
Load	s		10		Opcode							Immediate									Dest.						Source1					
Store	s		10		Opcode							Immediate									Source2						Source1					
CondBranch	s		11		1	Opcode			BrReg			Immediate																				
CntlTransfer	s		11		0	Opcode						Immediate																				

Figure 3: 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** (INT3-Reg) and **Immediate** (INT3-Im). Bit 27 specifies the INT3 format, 0=**Register** format, 1=**Immediate**. In **Register** format, the operation consists of **Dest = Src1 Op Src2**. **Immediate** format consists of **Dest = 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 (**BrDest**). 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 **Dest=Mem[Src1+Immediate]** semantics, while stores follow **Mem[Src1+Immediate]=Src2**. Thus bits 6 to 11 specify either the target destination register (**Dest**) or the second operand source register (**Src2**), depending on whether a load or store operation, respectively.

A.4 Opcodes

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
add	s		00		00		00000								dest						src2						src1					
sub	s		00		00		00001								dest						src2						src1					
shl	s		00		00		00010								dest						src2						src1					
shr	s		00		00		00011								dest						src2						src1					
shru	s		00		00		00100								dest						src2						src1					
sh1add	s		00		00		00101								dest						src2						src1					

Figure 4: Instruction Encodings

<table>
<tr><th></th><th>31</th><th>30</th><th>29 28</th><th>27 26</th><th>25</th><th>24 23 22 21</th><th>20 19 18</th><th>17 16 15 14 13 12</th><th>11 10 9 8 7 6</th><th>5 4 3 2 1 0</th></tr>
<tr><td>sh2add</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">00110</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>sh3add</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">00111</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>sh4add</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">01000</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>and</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">01001</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>andc</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">01010</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>or</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">01011</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>orc</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">01100</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>xor</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">01101</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>sxtb</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">01110</td><td></td><td>dest</td><td>src2</td><td></td></tr>
<tr><td>sxth</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">01111</td><td></td><td>dest</td><td>src2</td><td></td></tr>
<tr><td>max</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">10000</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>maxu</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">10001</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>min</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">10010</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>minu</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">10011</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>bswap</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">10100</td><td></td><td>dest</td><td>src2</td><td></td></tr>
<tr><td>mull</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">10101</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>mullu</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">10110</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>mulh</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">10111</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>mulhu</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">11000</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>mulll</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">11001</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>mulllu</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">11010</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>mullh</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">11011</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>mullhu</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">11100</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>mulhh</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">11101</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>mulhhu</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">11110</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>mulhs</td><td>s</td><td></td><td>00</td><td>00</td><td colspan="2">11111</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>cmpeq</td><td>s</td><td></td><td>00</td><td colspan="2">010</td><td>0000</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>cmpne</td><td>s</td><td></td><td>00</td><td colspan="2">010</td><td>0001</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>cmpge</td><td>s</td><td></td><td>00</td><td colspan="2">010</td><td>0010</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>cmpgeu</td><td>s</td><td></td><td>00</td><td colspan="2">010</td><td>0011</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>cmpgt</td><td>s</td><td></td><td>00</td><td colspan="2">010</td><td>0100</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>cmpgtu</td><td>s</td><td></td><td>00</td><td colspan="2">010</td><td>0101</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>cmple</td><td>s</td><td></td><td>00</td><td colspan="2">010</td><td>0110</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>cmpleu</td><td>s</td><td></td><td>00</td><td colspan="2">010</td><td>0111</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>cmplt</td><td>s</td><td></td><td>00</td><td colspan="2">010</td><td>1000</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>cmpltu</td><td>s</td><td></td><td>00</td><td colspan="2">010</td><td>1001</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
<tr><td>andl</td><td>s</td><td></td><td>00</td><td colspan="2">010</td><td>1010</td><td></td><td>dest</td><td>src2</td><td>src1</td></tr>
</table>

Figure 4: 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	010	1011		dest	src2	src1
orl	s		00	010	1100		dest	src2	src1
norl	s		00	010	1101		dest	src2	src1
cmpeq	s		00	011	0000	bdest		src2	src1
cmpne	s		00	011	0001	bdest		src2	src1
cmpge	s		00	011	0010	bdest		src2	src1
cmpgeu	s		00	011	0011	bdest		src2	src1
cmpgt	s		00	011	0100	bdest		src2	src1
cmpgtu	s		00	011	0101	bdest		src2	src1
cmple	s		00	011	0110	bdest		src2	src1
cmpleu	s		00	011	0111	bdest		src2	src1
cmplt	s		00	011	1000	bdest		src2	src1
cmpltu	s		00	011	1001	bdest		src2	src1
andl	s		00	011	1010	bdest		src2	src1
nandl	s		00	011	1011	bdest		src2	src1
orl	s		00	011	1100	bdest		src2	src1
norl	s		00	011	1101	bdest		src2	src1
add	s		00	10	00000		isrc2	idest	src1
sub	s		00	10	00001		isrc2	idest	src1
shl	s		00	10	00010		isrc2	idest	src1
shr	s		00	10	00011		isrc2	idest	src1
shru	s		00	10	00100		isrc2	idest	src1
sh1add	s		00	10	00101		isrc2	idest	src1
sh2add	s		00	10	00110		isrc2	idest	src1
sh3add	s		00	10	00111		isrc2	idest	src1
sh4add	s		00	10	01000		isrc2	idest	src1
and	s		00	10	01001		isrc2	idest	src1
andc	s		00	10	01010		isrc2	idest	src1
or	s		00	10	01011		isrc2	idest	src1
orc	s		00	10	01100		isrc2	idest	src1
xor	s		00	10	01101		isrc2	idest	src1
max	s		00	10	10000		isrc2	idest	src1
maxu	s		00	10	10001		isrc2	idest	src1
min	s		00	10	10010		isrc2	idest	src1
minu	s		00	10	10011		isrc2	idest	src1
mull	s		00	10	10101		isrc2	idest	src1
mullu	s		00	10	10110		isrc2	idest	src1

Figure 4: 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
mulh	s		00	10	10111	isrc2	idest	src1
mulhu	s		00	10	11000	isrc2	idest	src1
mulll	s		00	10	11001	isrc2	idest	src1
mullu	s		00	10	11010	isrc2	idest	src1
mullh	s		00	10	11011	isrc2	idest	src1
mullhu	s		00	10	11100	isrc2	idest	src1
mulhh	s		00	10	11101	isrc2	idest	src1
mulhhu	s		00	10	11110	isrc2	idest	src1
mulhs	s		00	10	11111	isrc2	idest	src1
cmpeq	s		00	110	0000	isrc2	idest	src1
cmpne	s		00	110	0001	isrc2	idest	src1
cmpge	s		00	110	0010	isrc2	idest	src1
cmpgeu	s		00	110	0011	isrc2	idest	src1
cmpgt	s		00	110	0100	isrc2	idest	src1
cmpgtu	s		00	110	0101	isrc2	idest	src1
cmple	s		00	110	0110	isrc2	idest	src1
cmpleu	s		00	110	0111	isrc2	idest	src1
cmplt	s		00	110	1000	isrc2	idest	src1
cmpltu	s		00	110	1001	isrc2	idest	src1
andl	s		00	110	1010	isrc2	idest	src1
nandl	s		00	110	1011	isrc2	idest	src1
orl	s		00	110	1100	isrc2	idest	src1
norl	s		00	110	1101	isrc2	idest	src1
cmpeq	s		00	111	0000	isrc2	ibdest	src1
cmpne	s		00	111	0001	isrc2	ibdest	src1
cmpge	s		00	111	0010	isrc2	ibdest	src1
cmpgeu	s		00	111	0011	isrc2	ibdest	src1
cmpgt	s		00	111	0100	isrc2	ibdest	src1
cmpgtu	s		00	111	0101	isrc2	ibdest	src1
cmple	s		00	111	0110	isrc2	ibdest	src1
cmpleu	s		00	111	0111	isrc2	ibdest	src1
cmplt	s		00	111	1000	isrc2	ibdest	src1
cmpltu	s		00	111	1001	isrc2	ibdest	src1
andl	s		00	111	1010	isrc2	ibdest	src1
nandl	s		00	111	1011	isrc2	ibdest	src1
orl	s		00	111	1100	isrc2	ibdest	src1
norl	s		00	111	1101	isrc2	ibdest	src1

Figure 4: 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
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		0101				0																							
immr	s		01		0101				1																							
slct	s		01		1	000			scond					isrc2									idest						src1			
slctf	s		01		1	001			scond					isrc2									idest						src1			
prgins	s		01			11111					00																					
sbrk	s		01			11111					01																					
syscall	s		01			11111					10																					
break	s		01			11111					11																					
ldw	s		10			0000000								isrc2									idest						src1			
ldw.d	s		10			0000100								isrc2									idest						src1			
ldh	s		10			0001000								isrc2									idest						src1			
ldh.d	s		10			0001100								isrc2									idest						src1			
ldhu	s		10			0010000								isrc2									idest						src1			
ldhu.d	s		10			0010100								isrc2									idest						src1			
ldb	s		10			0011000								isrc2									idest						src1			
ldb.d	s		10			0011100								isrc2									idest						src1			
ldbu	s		10			0100000								isrc2									idest						src1			
ldbu.d	s		10			0100100								isrc2									idest						src1			
stw	s		10			0101000								isrc2									src2						src1			
sth	s		10			0101100								isrc2									src2						src1			
stb	s		10			0110000								isrc2									src2						src1			
pft	s		10			0110100								isrc2															src1			
prgadd	s		10			0111000								isrc2															src1			
prgset	s		10			0111100								isrc2															src1			
sync	s		10			1000000																										
call	s		11		0	0000																										
call	s		11		0	0001																										
goto	s		11		0	0010																										
goto	s		11		0	0011																										
rfi	s		11		0	0100																										
br	s		11		1	0	bcond																									
brf	s		11		1	1	bcond																									

Figure 4: Instruction Encodings

Index