Xtensa®
Instruction Set Architecture (ISA)

Reference Manual

For All Xtensa Processor Cores

Tensilica, Inc.
3255-6 Scott Blvd.
Santa Clara, CA 95054
(408) 986-8000
fax (408) 986-8919
www.tensilica.com

This publication is provided “AS IS.” Tensilica, Inc. (hereafter “Tensilica”) does not make any warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Information in this document is provided solely to enable system and software developers to use Tensilica processors. Unless specifically set forth herein, there are no express or implied patent, copyright or any other intellectual property rights or licenses granted hereunder to design or fabricate Tensilica integrated circuits or integrated circuits based on the information in this document. Tensilica does not warrant that the contents of this publication, whether individually or as one or more groups, meets your requirements or that the publication is error-free. This publication could include technical inaccuracies or typographical errors. Changes may be made to the information herein, and these changes may be incorporated in new editions of this publication.

Issue Date: 4/2010
RC-2010.1 Release
PD-09-0801-10-01

Tensilica, Inc.
3255-6 Scott Blvd.
Santa Clara, CA 95054
(408) 986-8000
fax (408) 986-8919
www.tensilica.com

Contents

List of Figures

List of Tables

Preface

This manual is written for Tensilica customers who are experienced in working with microprocessors or in writing assembly code or compilers. It is NOT a specification for one particular implementation of the Architecture, but rather a reference for the ongoing Instruction Set Architecture. For a detailed specification for specific products, refer to a specific Tensilica processor data book.

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- `$` represents your shell prompt, in user-session examples.
- **`literal_input`** indicates literal command-line input.
- `variable` indicates a user parameter.
- `literal_keyword (`in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- `... output ...` indicates unspecified program output.
- `[optional-variable]` indicates an optional parameter.
- **[**`variable`**]** indicates a parameter within literal square-braces.
- **{**`variable`**}** indicates a parameter within literal curly-braces.
- **(**`variable`**)** indicates a parameter within literal parentheses.
- | means *OR*.
- `(var1 | var2)` indicates a required choice between one of multiple parameters.
- `[var1 | var2]` indicates an optional choice between one of multiple parameters.
- `var1 [, varn]*` indicates a list of 1 or more parameters (0 or more repetitions).
- `4'b0010` is a 4-bit value specified in binary.
- `12'o7016` is a 12-bit value specified in octal.
- `10'd4839` is a 10-bit value specified in decimal.
- `32'hff2a` or `32'HFF2A` is a 32-bit value specified in hexadecimal.

Terms

- *0x* at the beginning of a value indicates a hexadecimal value.
- *b* means bit.
- *B* means byte.

- *flush* is deprecated due to potential ambiguity (it may mean *write-back* or *discard)*.
- *Mb* means megabit.
- *MB* means megabyte.
- *PC* means program counter.
- *word* means 4 bytes.

Related Tensilica Documents

- *330HiFi Standard DSP Processor Data Book*
- *388VDO Hardware User's Guide*
- *388VDO Software Guide*
- *545CK Standard DSP Processor Data Book*
- *ConnX D2 DSP Engine User's Guide*
- *ConnX Vectra™ LX DSP Engine Guide*
- *Diamond Series Hardware User's Guide*
- *Diamond Series Upgrade Guide*
- *Diamond Standard Controllers Data Book*
- *GNU Assembler User's Guide*
- *GNU Binary Utilities User's Guide*
- *GNU Debugger User's Guide*
- *GNU Linker User's Guide*
- *GNU Profiler User's Guide*
- *HiFi 2 Audio Engine Codecs Programmer's Guides*
- *HiFi 2 Audio Engine Instruction Set Architecture Reference Manual*
- *Red Hat newlib C Library Reference Manual*
- *Red Hat newlib C Math Library Reference Manual*
- *Tensilica Avnet LX200 (XT-AV200) Board User's Guide*
- *Tensilica Avnet LX60 (XT-AV60) Board User's Guide*
- *Tensilica Bus Designer's Toolkit Guide*
- *Tensilica C Application Programmer's Guide*
- *Tensilica Instruction Extension (TIE) Language Reference Manual*
- *Tensilica Instruction Extension (TIE) Language User's Guide*
- *Tensilica On-Chip Debugging Guide*
- *Tensilica Processors Bus Bridges Guide*
- *Tensilica Trace Solutions User's Guide*
- *Xtensa® C and C++ Compiler User's Guide*
- *Xtensa® Development Tools Installation Guide*
- *Xtensa® Energy Estimator (Xenergy) User's Guide*
- *Xtensa® Hardware User's Guide*
- *Xtensa® Instruction Set Architecture (ISA) Reference Manual*
- *Xtensa® Instruction Set Simulator (ISS) User's Guide*
- *Xtensa® Linker Support Packages (LSPs) Reference Manual*

- *Xtensa® LX3 Microprocessor Data Book*
- *Xtensa® 8 Microprocessor Data Book*
- *Xtensa® Microprocessor Programmer’s Guide*
- *Xtensa® Modeling Protocol (XTMP) User’s Guide*
- *Xtensa® OSKit™ Guide*
- *Xtensa® Processor Extensions Synthesis (XPRES™) Compiler User’s Guide*
- *Xtensa® Processor Interface Protocol Reference Manual*
- *Xtensa® Software Development Toolkit User's Guide*
- *Xtensa® SystemC® (XTSC) Reference Manual*
- *Xtensa® SystemC® (XTSC) User’s Guide*
- *Xtensa® System Designer’s Guide*
- *Xtensa® System Software Reference Manual*
- *Xtensa® Upgrade Guide*

Changes from the Previous Version

The following changes have been made to this document for the Tensilica RC-2010.1 release:

- Deleted several extraneous blank pages in between each chapter in previous release.
- Corrected erroneous cross-references to Table 4–55 through Table 4–58 in Section 4.4.1.1 on page 83
- Clarified information about lookup rings in Section 4.6.2.2 and Section 4.6.2.3.

The following changes have been made to this document for the Tensilica RC-2009.0 release:

- A new register, ATOMCL, has been added to Section 4.3.13 “Conditional Store Option” on page 91. The ATOMCTL register controls the interaction of the S32C1I instruction with the memory system.
- The description of attributes for the Section 4.6.3 “Region Protection Option” on page 187 and the Section 4.6.5.10 “MMU Option Memory Attributes” on page 213 have been improved. There are no actual changes to the attributes.
- The Section 4.6.5 “MMU Option” on page 196 has gained a new option. Way5 and Way6 can now be either variable or fixed. The variable version provides more flexibility in the address map and has a setting where the MMU puts out a physical address equal to the virtual address and is, in that sense, turned off.
- Many of the SYNC instruction requirements listed in Section 5.3 “Special Registers” on page 259 have not actually been needed after T1050. Those requirements have now been removed from Section 5.3 but retained in Appendix A.
- The RER and WER instructions have been added to Chapter 6.

1. Introduction

This chapter provides an overview of Tensilica, the Xtensa Instruction Set Architecture (ISA), and the Xtensa Processor Generator.

1.1 What Problem is Tensilica Solving?

Processors have traditionally been extremely difficult to design and modify. Therefore, most systems contain rigid processors that were designed and verified once for general-purpose use and then embedded into multiple applications over time. Because these processors are general-purpose designs, their suitability to any particular application is less than ideal. Although it would be preferable to have a processor specifically designed to execute a particular application's code better (for example, to run faster, or consume less power, or cost less), this is rarely possible because of the difficulty; the time, cost, and risk of modifying an existing processor or developing a new processor is very high.

It is also not appropriate to simply design traditional processors with more features to cover all applications, because any given application only requires a particular set of features — a processor with features not required by the application is overly costly and consumes unnecessary power. It is also not possible to know all of the potential application targets when a processor is initially designed.

If processor configuration could be automated and made reliable, then system designers would have the option and ability to create truly efficient application solutions.

This is just what Tensilica is about: Tensilica provides a set of techniques and tools for designing an application solution that contains one or more processors, each one configured and enhanced at design-time to fine-tune its suitability for a specific application. Fine-tuning an architecture can consist of any combination of:

- *Extensibility*: Adding architectural enhancements.
- *Configurability*: Creating custom processor configurations.
- *Retargetability*: Mapping the architecture into hardware to meet different speed, area, and power targets in different processes.

1.1.1 Adding Architectural Enhancements

As an example of an architectural enhancement, consider a device designed to transmit and receive data over a channel using a complex protocol. Because the protocol is complex, the processing cannot be reasonably accomplished entirely in hard logic, and instead a programmable processor is introduced into the system for protocol processing. This processor's programmability also allows bug fixes and upgrades to later protocols to be done by loading the instruction memories with new software. However, the processor was probably not designed for this particular application (the application may not have even existed when the processor was designed), and the application may perform operations that require many instructions — operations that could be accomplished with a trivial amount of additional processor logic.

Before the introduction of Tensilica's Xtensa technology, processors could not be enhanced easily. Because of this, many system designers are forced to solve problems by executing the inefficient pure-software solution on the available general-purpose processor. This results in a solution that may be slower, or higher power, or costlier than necessary (for example, it may require a larger, more powerful processor to execute the program at sufficient speed).

Other designers choose to provide some of the processing requirements in special-purpose hardware that they design for the application. This approach requires special code to access the custom hardware at various points in the program. However, the time to transfer data between the processor and the custom hardware limits the utility of this approach to fairly large units of work; small computations cannot sufficiently amortize the communication overhead introduced by this approach to provide a reasonable speed-up.

In the communication-channel application example, the protocol might require encryption, error-correction, or compression/decompression processing. Such processing often operates on individual bits rather than a processor's larger words. The circuitry for a computation may be rather modest, but the need for the processor to extract each bit, sequentially process it, and then repack the bits adds considerable overhead.

As a specific example, consider the Huffman decode shown in Table 1–1.

Table 1–1. Huffman Decode Example

Input	Value	Length
`00xxxxxx`	0	2
`01xxxxxx`	1	2
`10xxxxxx`	2	2
`110xxxxx`	3	3
`1110xxxx`	4	4
`11110xxx`	5	5
`111110xx`	6	6
`1111110x`	7	7
`11111110`	8	8
`11111111`	9	8

Both the value and the length must be computed, so that length bits can be shifted off to find the start of the next token. (A similar encoding is used in the MPEG compression standard.) There are many ways to code this for a conventional RISC instruction set, but all of them require many instructions, because there are many tests to be done, and each test requires a single cycle (as opposed to a single gate delay for logic). For example, in the MIPS instruction set, the above decode procedure might look like this:

```
/* input in t0, value out in t1, length out in t2 */
     srl  t1, t0, 6
     li   t3, 3
     beq  t3, t4, 2f
      li  t2, 2
     andi t3, t0, 0x20
     beq  t3, r0, 1f
      li  t2, 3
     andi t3, t0, 0x10
     beq  t3, r0, 1f
      li  t2, 4
     andi t3, t0, 0x08
     beq  t3, r0, 1f
      li  t2, 5
     andi t3, t0, 0x04
     beq  t3, r0, 1f
      li  t2, 6
     andi t3, t0, 0x02
     beq  t3, r0, 1f
      li  t2, 7
     andi t3, t0, 0x01
     beq  t3, r0, 1f
      li  t2, 8
     b    2f
      li  t1, 9
1:   /* length = value */
     move t1, t2
2:   /* done */
```

This is so expensive that a 256-entry lookup table is typically used instead. However, a 256-entry lookup table takes significant space and can take many cycles to access. For longer Huffman encodings, the table size would become prohibitive, leading to more complex and slower code.

The logic to decode this requires roughly 30 gates (just the combinatorial logic function, not counting instruction decode and so forth) — less than 0.1% of a processor gate-count — and can be computed by a special-purpose processor instruction in a single cycle. This is a factor of 4 to 20 speed-up over using general-purpose instructions only. A processor extended to have this logic in the form of an instruction would simply do:

```
huff8t1, t0    /* t1[3:0] is length, t1[7:0] is value */
```

Tensilica’s solution is to provide a mechanism with which to easily and efficiently extend processor architecture with application-specific instructions.

1.1.2 Creating Custom Processor Configurations

While the ability to extend processor architecture, which we call *extensibility*, lets system designers incorporate new functionality into a processor, *configurability* lets processor designers specify whether (or how much) pre-designed functionality is required for a particular product.

The simplest sort of configurability is a binary choice: an architectural feature is either present or absent in a particular processor configuration. For example, a processor might be offered either with or without floating-point hardware. Multiple configurations of a set of architectural features could be created by the processor designer, not the system designer.

System-design flexibility is improved by having finer gradations in processor-configuration choices. For example, a processor configuration might allow the system designer to specify the number of registers in the register file, memory width, cache size, cache associativity, and so on.

1.1.3 Mapping the Architecture into Hardware

Extensibility and configurability provide great flexibility. However, the resulting design must still be mapped into physical hardware. Synthesis, placement, and routing tools allow high-level representations of a design to be automatically mapped into more detailed designs. While these mapping operations do not change the functionality of the design, they are important building blocks that facilitate extensibility and configurability.

Many processors are manually designed all the way to the layout. For such a processor design, extensibility and configurability would require changes to the layout. By contrast, the Tensilica system builds on existing synthesis, placement, and routing tools so that configuration need only change the input to synthesis, and conventional mapping techniques are used to create physical hardware.

Some synthesis tools choose different mapping based on the designer’s goal specifications, allowing the mapping to optimize for speed, power, area, or target components. This is as close to providing configurability that existing mapping tools come: the designer can specify different synthesis parameters for a fixed input. By contrast, the Tensilica approach lets the designer alter the input to synthesis, and change its functionality.

1.1.4 Development and Verification Tools

Extending an architecture and reconfiguring a processor may require widespread changes in processor logic to keep pipeline stages synchronized. Such reconfiguration requires that the processor be re-verified. Tensilica automates these changes and makes them reliable.

In addition, when the processor changes, the software tool chain — compilers, assemblers, linkers, debuggers, simulators, and profilers — must change as well. In the past, the cost of software changes associated with processor reconfigurations has been a major impediment. Tensilica automates these changes also.

Finally, it should be possible to get feedback on the performance, cost, power, and other effects of processor reconfiguration without taking the design through the entire mapping process. This feedback can be used to direct further reconfiguration of the processor until the system design goals are achieved. Tensilica's technology dramatically improves the feedback loop.

1.2 The Xtensa Instruction Set Architecture

The Xtensa Instruction Set Architecture (ISA) is a new post-RISC ISA targeted at embedded, communication, and consumer products. The ISA is designed to provide:

- A high degree of extensibility
- Industry-leading code density
- Optimized low-power implementation
- High performance
- Low-cost implementation

This manual describes the Xtensa ISA — both the core architecture and the architectural options. Figure 1–1 illustrates the general organization of the processor hardware in which the Xtensa ISA is implemented. This manual does not describe the memory map, protection model, or peripherals that can be implemented in particular configurations of the Xtensa ISA.

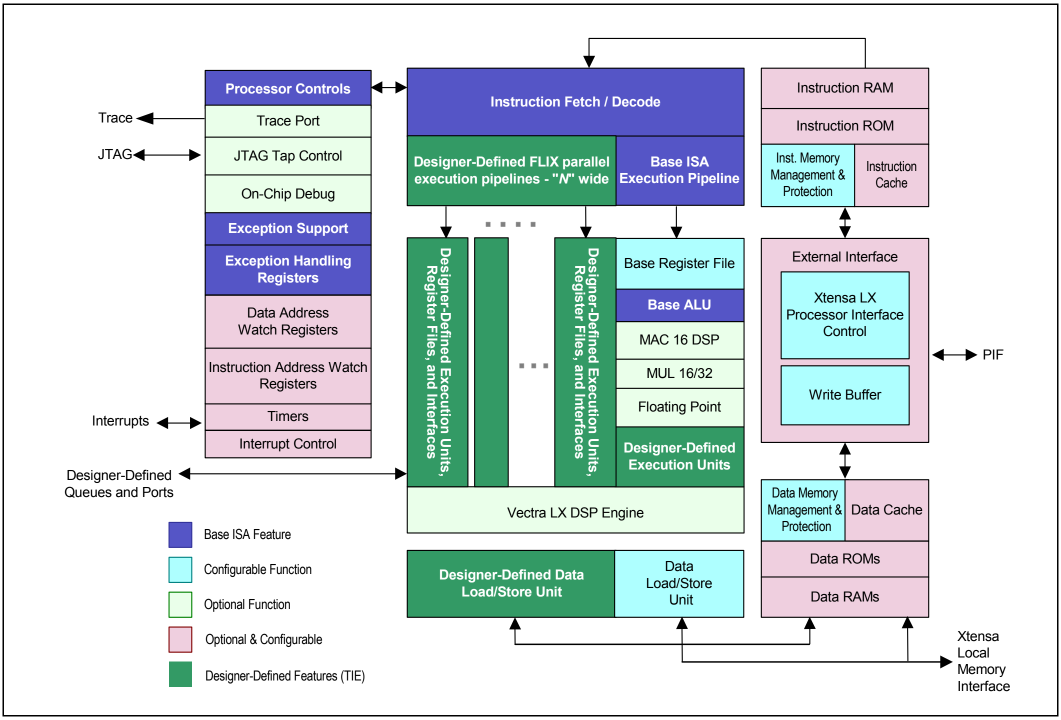

Trace
JTAG
Interrupts
Designer-Defined Queues and Ports
Processor Controls
Trace Port
JTAG Tap Control
On-Chip Debug
Exception Support
Exception Handling Registers
Data Address Watch Registers
Instruction Address Watch Registers
Timers
Interrupt Control
Instruction Fetch / Decode
Designer-Defined FLIX parallel execution pipelines - "N" wide
Base ISA Execution Pipeline
Designer-Defined Execution Units, Register Files, and Interfaces
Designer-Defined Execution Units, Register Files, and Interfaces
Base Register File
Base ALU
MAC 16 DSP
MUL 16/32
Floating Point
Designer-Defined Execution Units
Vectra LX DSP Engine
Designer-Defined Data Load/Store Unit
Data Load/Store Unit
Instruction RAM
Instruction ROM
Inst. Memory Management & Protection
Instruction Cache
External Interface
Xtensa LX Processor Interface Control
Write Buffer
PIF
Data Memory Management & Protection
Data Cache
Data ROMs
Data RAMs
Xtensa Local Memory Interface
Base ISA Feature
Configurable Function
Optional Function
Optional & Configurable
Designer-Defined Features (TIE)

Figure 1–1. Xtensa LX Hardware Architecture Block Diagram

Table 1–2 compares the architectural features provided by the Xtensa ISA to those of typical RISC architectures. Each of the Xtensa features are described in this manual.

Table 1–2. Comparison of Typical RISC and Xtensa ISA Features

Architectural Feature	Typical RISC	Xtensa
Instruction size	32 bits	24 and 16 bit
Compare and branch	no or partial	total
Application-specific instructions	no	yes
Zero-overhead loop	no	yes
Funnel shift	no (except 29000)	yes
Variable-increment register windows	no	yes
Conditional move	recently	yes
Compound multiply/add	recently	yes
Advanced multiprocessor synchronization	recently	yes

1.2.1 Configurability

The Xtensa ISA goes further than incorporating post-RISC features: it is modular, consisting of a core architecture and architectural options. Table 1–3 lists the initial set of modular components.

Table 1–3. Modular Components

Component	Reference
Core Architecture	Chapter 3, "Core Architecture" on page 23
Core Architecture	Section 4.2 “Core Architecture” on page 50
Options for Additional Instructions	
Code Density Option	"Code Density Option" on page 53
Loop Option	"Loop Option" on page 54
Extended L32R Option	"Extended L32R Option" on page 56
16-bit Integer Multiply Option	"16-bit Integer Multiply Option" on page 57
32-bit Integer Multiply Option	"32-bit Integer Multiply Option" on page 58
MAC16 Option	"MAC16 Option" on page 60
Miscellaneous Operations Option	"Miscellaneous Operations Option" on page 62
Coprocessor Option	"Coprocessor Option" on page 63
Boolean Option	"Boolean Option" on page 65
Floating-Point Coprocessor Option	"Floating-Point Coprocessor Option" on page 67
Multiprocessor Synchronization Option	"Multiprocessor Synchronization Option" on page 74
Conditional Store Option	"Conditional Store Option" on page 77
Options for Interrupts and Exceptions	
Exception Option	"Exception Option" on page 82
Unaligned Exception Option	"Unaligned Exception Option" on page 99
Interrupt Option	"Interrupt Option" on page 100
High-Priority Interrupt Option	"High-Priority Interrupt Option" on page 106
Timer Interrupt Option	"Timer Interrupt Option" on page 110

Table 1–3. Modular Components (continued)

Component	**Reference**
Options for Memory	
Instruction Cache Option	"Instruction Cache Option" on page 115
Instruction Cache Test Option	"Instruction Cache Test Option" on page 116
Instruction Cache Index Lock Option	"Instruction Cache Index Lock Option" on page 117
Data Cache Option	"Data Cache Option" on page 118
Data Cache Test Option	"Data Cache Test Option" on page 121
Data Cache Index Lock Option	"Data Cache Index Lock Option" on page 122
Instruction RAM Option	"Instruction RAM Option" on page 124
Instruction ROM Option	"Instruction ROM Option" on page 125
Data RAM Option	"Data RAM Option" on page 126
Data ROM Option	"Data ROM Option" on page 126
XLMI Option	"XLMI Option" on page 127
Hardware Alignment Option	"Hardware Alignment Option" on page 128
Memory ECC/Parity Option	"Memory ECC/Parity Option" on page 128
Options for Memory Protection	
Region Protection Option	"Region Protection Option" on page 150
Region Translation Option	"Region Translation Option" on page 156
MMU Option	"MMU Option" on page 158
Options for Other Purposes	
Windowed Register Option	"Windowed Register Option" on page 180
Processor Interface Option	"Processor Interface Option" on page 194
Miscellaneous Special Registers Option	"Miscellaneous Special Registers Option" on page 195
Thread Pointer Option	"Thread Pointer Option" on page 196
Processor ID Option	"Processor ID Option" on page 196
Debug Option	"Debug Option" on page 197
Trace Port Option	"Trace Port Option" on page 203

1.2.2 Extensibility

In addition to the Xtensa components shown in Table 1–3, designers can extend the Xtensa architecture by adding States, Register Files, and instructions that operate both on the AR Register File and on the additional states the designer has added. These instructions can be single cycle or multiple cycles, and share or re-use logic.

1.2.2.1 State Extensions

The designer can add State Registers. These State Registers can be the source or destination of various instructions and are saved and restored by the operating system.

1.2.2.2 Register File Extensions

The designer can add Register Files of widely varying size. These Register Files can be the source or destination of various instructions and are saved and restored by the operating system. The registers within them are allocated by the compiler, which can spill and re-fill them if necessary.

1.2.2.3 Instruction Extensions

The designer can define new instructions that contain simple functions consisting of combinatorial logic that takes one or two source operands from registers and produces a result to be written to a register:

```
AR[r] ← f(AR[s], AR[t])
```

Instructions can also be much more complex with register file values and State appearing as both inputs and outputs. These Instructions are described using the Tensilica Instruction Extension (TIE) language (see Section 1.3.2).

1.2.2.4 Coprocessor Extensions

Another mechanism to extend the Xtensa ISA is to use the Coprocessor Option. A coprocessor is defined as a combination of registers, other state, and logic that operates on that state, including loads, stores and setting of Booleans for branch true/false operations. A particular coprocessor can be enabled or disabled to control with one bit whether or not instructions accessing that combination of registers and other state may or may not execute.

1.2.3 Time-to-Market

The Xtensa Software Development Toolkit includes automatically generated software that matches the designer’s processor configuration and eliminates tool headaches. The ISA’s rich set of features (for example, interrupt and debug facilities) makes the system designer’s job easier. The ability to create custom instructions with the TIE language allows the designer to reach performance goals with less code-tuning or hard-to-interface-to external logic.

1.2.4 Code Density

The Xtensa core ISA is implemented as 24-bit instructions. This instruction width provides a direct 25% reduction in code size compared with 32-bit ISAs. The instructions provide access to the entire processor hardware and support special functions, such as single-instruction compare-and-branch, which reduce the number of instructions required to implement various applications. These special functions result in further code-size reductions.

The Xtensa ISA also includes a Code Density Option that further reduces code size. This option adds 16-bit instructions that are distinguished by opcode, and that can be freely intermixed with 24-bit instructions to achieve higher code density than competing ISAs without giving up the performance of a 32-bit ISA. The 16-bit instructions add no new functionality but provide compact encoding of the most frequently used 24-bit instructions. In typical code, roughly half of all instructions can be encoded in 16 bits.

The core ISA omits the branch delay slots required by some RISC ISAs. This increases code density by eliminating NOPs the compiler uses to fill the slot after a branch when it cannot find a real instruction to put there (only 50% of the branch delay slots are filled on some RISC architectures).

The Xtensa ISA provides a Windowed Registers Option. Xtensa windowed registers reduce code size by:

- Eliminating register saves and restores at procedure entry and exit
- Reducing argument shuffling
- Allowing more local variables to live permanently in registers

1.2.5 Low Implementation Cost

The Xtensa architecture is designed to facilitate efficient implementation. It can be implemented with simple instruction pipelines and direct hardware execution without micro code. Operations that are too complex to easily implement with single instructions are synthesized into appropriate instruction sequences by the compiler. The base architecture avoids instructions that would need extra register file read or write ports. This keeps the minimal configuration low-cost and low-power.

The Xtensa architecture fully supports the common data types and operations found in a broad range of applications. The base architecture omits special-purpose data types and operations. Optional instructions, the TIE language (see Section 1.3.2), and optional coprocessors allow the designer to add exactly the functionality needed, thus reducing the cost and performance due to unused general-purpose functions.

The Xtensa ISA's improvements in code size help reduce system cost (for example, by reducing the amount of ROM, Flash, or RAM required). Making features like the number of debug registers configurable allows the system designer, instead of the processor designer, to decide the cost/benefit trade-off.

1.2.6 Low-Power

The Xtensa ISA has several energy-efficient attributes that enhance battery-operated systems. The core ISA is built on 32-bit operations; some embedded processors of similar performance have 64-bit base operations, which consumes additional power, often unnecessarily. (TIE does allow 64-bit or greater computations to be added to the processor for those algorithms that require it, but these can be used selectively to achieve a balance between performance and power consumption.)

The core ISA uses a register file with only two read ports and one write port, a configuration that requires fewer transistors and less power than architectures with more ports.

The Xtensa Windowed Registers Option saves power by reducing the number of dynamic data-memory references and increasing the opportunities for variables to reside in registers, where accesses require less power than memory accesses.

The `WAITI` (Wait for Interrupt) instruction, which is a part of the Interrupt Option, saves power by setting the current interrupt level, powering down the processor's logic, and waiting for an interrupt.

1.2.7 Performance

The Xtensa ISA achieves its extensibility, code density, and low-power advantages without sacrificing performance. For example, the Thumb and MIPS16 extensions of the ARM and MIPS ISAs, respectively, provide improved code density by using only eight registers and by reducing operand flexibility. By contrast, the Xtensa 24-bit instructions can access 16 virtual registers with 3 register operands, and 16-bit instructions can access all 16 registers with 1 to 3 register operands. The mapping of the 16 virtual registers to the physical register file can eliminate register saves and restores at procedure entry and exit, also increasing performance.

The Xtensa ISA also enhances performance by providing:

- A complete set of compare-and-branch instructions, eliminating the need for separate comparison instructions
- `LOOP`, `LOOPNEZ`, and `LOOPGTZ` instructions that provide zero-overhead looping

These features are described in Section 3.8 of this manual. Other features of the architecture minimize critical paths, allow better compiler scheduling, and require fewer executed instructions to implement a given program.

1.2.8 Pipelines

The Xtensa ISA can be implemented using a variety of pipelines. A 5-stage load-store oriented pipeline, such as is used in many RISC processors, is supported by Xtensa implementations and illustrated in Figure 1–2. Many other variations are possible. A 7-stage load-store oriented pipeline is supported by some Xtensa implementations. Instructions can also have computation in later pipe stages so that the computation can use memory data loaded by the same instruction.

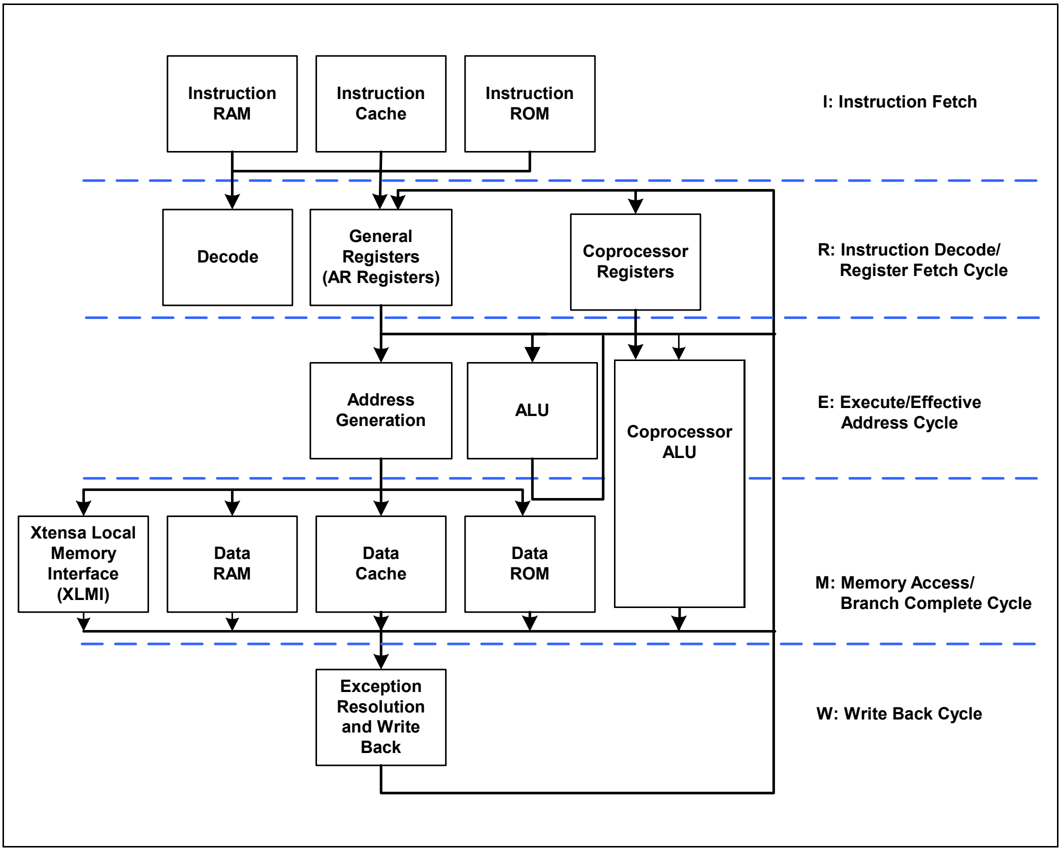

Instruction RAM
Instruction Cache
Instruction ROM
I: Instruction Fetch
Decode
General Registers (AR Registers)
Coprocessor Registers
R: Instruction Decode/ Register Fetch Cycle
Address Generation
ALU
Coprocessor ALU
E: Execute/Effective Address Cycle
Xtensa Local Memory Interface (XLMI)
Data RAM
Data Cache
Data ROM
M: Memory Access/ Branch Complete Cycle
Exception Resolution and Write Back
W: Write Back Cycle

Figure 1–2. Example Implementation Pipeline

The instruction set was also designed with a 2-read, 1-write general register file (called Address Registers) in mind. While this approach results in lower implementation cost, it prevents the inclusion of auto-incrementing loads and indexed stores to or from the Address Registers. For the sake of symmetry, the ISA therefore does not include auto-incrementing stores and indexed loads. However, all of these addressing modes are possible for designer defined loads and stores. Designers can implement register files with more read and write ports. For example, the Xtensa Floating-Point Coprocessor Option contains a floating point register file with three read ports.

1.3 The Xtensa Processor Generator

The Xtensa Processor Generator is the key to rapid, optimal creation of application-specific processors. Using this tool, the designer can specify and generate a complete processor subsystem. The designer can select the instruction set, memory hierarchy, peripherals and interface options to fit the target application.

The Generator user interface captures designer input in several ways, including:

- Configuration of the processor micro-architecture
- Configuration of Tensilica-provided instruction and coprocessor options
- Specification of designer-defined instruction and coprocessor extensions, using the Tensilica Instruction Extension (TIE) language

Together, these specifications make up the configuration database shown near the top of Figure 1–3. This file is used to generate all the software tools and hardware descriptions for the final application-specific processor.

1.3.1 Processor Configuration

The Generator interface drives the creation and optimization of all forms of the processor needed for integration into the system design flow. Based on the designer’s specifications, it creates synthesizable Verilog or VHDL code, synthesis scripts, an HDL test bench, and physical placement files. Simultaneously, an optimized C and C++ compiler, assembler, linker, symbolic debugger, Instruction Set Simulator, libraries and verification tests are built for the designer’s software development.

The Generator interface lets the designer specify implementation targets for speed, area and process technology, as well as the optimization priorities used in synthesis and layout.

1.3.2 System-Specific Instructions—The TIE Language

The Tensilica Instruction Extension (TIE) language lets the designer add instructions to the processor implementation, including full software support for generated instructions. The specification of instruction extensions can include the following aspects as well as many others:

- Instruction Operation — Defines the operation of an additional instruction

- Immediate and Constant Tables — Defines constant values in instructions
- Register File — Defines new register files
- State — Defines new single processor states for instructions to operate on
- Length and Format — The FLIX extensions to TIE allow for multiple instruction sizes and the defining of multiple operations in a single instruction
- Queues and Ports — Defines input and output queue ports and other ports for the Xtensa processor
- Types — Defines new C/C++ data types associated with user defined register files. Allows type checking and automatic loading, storing and register allocation
- Prototypes — Defines the argument types of C/C++ intrinsics for each instruction and the instruction sequences for loading, storing, and moving the added types
- Schedule — Defines the pipeline stages at which instructions use input values and produce output values

In addition to designer-defined register and register file operands, instructions can use `AR` registers as source values. They may generate multiple results, including `AR` register file results. These instructions should be designed to have circuit delays appropriate to the number cycles specified in the `schedule` specifications to avoid limiting the processor clock frequency. The instruction semantics are expressed in a subset of Verilog, including all commonly used operators (multiply, add, subtract, minus, not, or, comparisons, reduction operators, shifts, concatenation, and conditionals).

The use of TIE for the creation of new instructions and coprocessors is described in the *Tensilica Instruction Extension (TIE) Language User's Guide*. The TIE language is described in the *Tensilica Instruction Extension (TIE) Language Reference Manual*.

Figure 1–3 illustrates the Xtensa design flow.

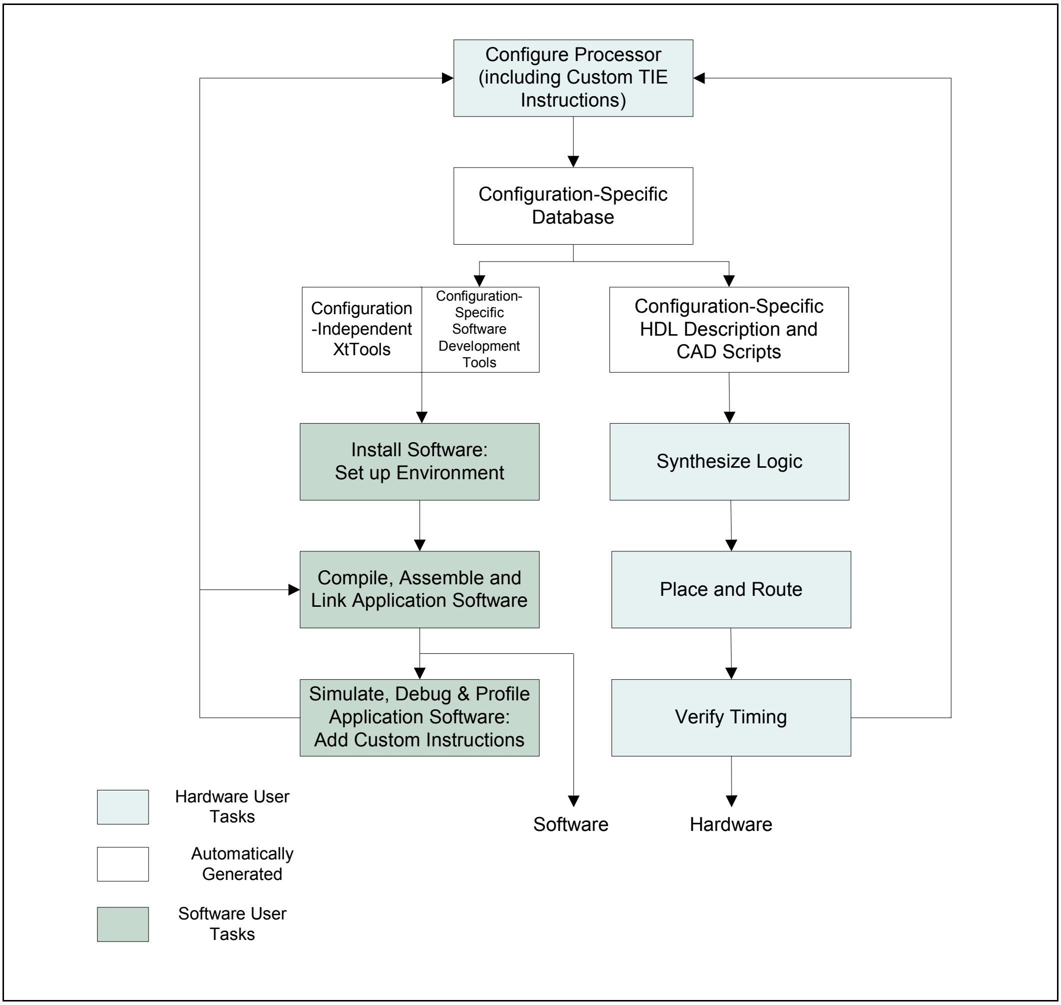

Configure Processor (including Custom TIE Instructions)
Configuration-Specific Database
Configuration -Independent XtTools
Configuration-Specific Software Development Tools
Configuration-Specific HDL Description and CAD Scripts
Install Software: Set up Environment
Synthesize Logic
Compile, Assemble and Link Application Software
Place and Route
Simulate, Debug & Profile Application Software: Add Custom Instructions
Verify Timing
Software
Hardware
Hardware User Tasks
Automatically Generated
Software User Tasks

Figure 1–3. The Xtensa Design Flow

2. Notation

This manual uses the following notation for instruction descriptions. Additional notation specific to opcode encodings is provided in "Opcode Encodings" on page 574.

2.1 Bit and Byte Order

This manual consistently uses little-endian bit ordering for describing instructions and registers. Bits in little-endian notation are numbered starting from 0 for the least-significant bit of a field. However, this notation convention is independent of how an Xtensa processor actually numbers bits, because a given processor can be configured for either little- or big-endian byte and bit ordering. For most Xtensa instructions, bit numbering is irrelevant; only the BBC and BBS instructions assign bit numbers to values on which the processor operates. The BBC/BBS instructions use big-endian bit ordering (0 is the most-significant bit) on a big-endian processor configuration. Bit numbering by the BBC/BBS instructions is illustrated in Figure 2–4.

In specifying little- or big-endian ordering during actual processor configuration, you are specifying both the bit and the byte order; the two orderings have the same most-significant and least-significant ends.

Figure 2–5 on page 18 illustrates big- and little-endian byte order, as implemented by Xtensa load (page 33) and store (page 36) instructions. Xtensa processors transfer data to and from the system using interfaces that are configurable in width (32, 64, or 128 bits in current implementations). These interfaces arrange their n bits according to their significance representing an n-bit unsigned integer value (that is, 0 to 2^n-1). Load and store instructions that reference quantities less than n bits access different bits of this integer in little-endian and big-endian byte orderings (for example, by changing the selection algorithm for loads). Xtensa processors *do not* rearrange bits of a word to implement endianness (for example, swapping bytes for big-endian operation).

Little-Endian bit numbering for BBC/BBS instructions:

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

←most-significant   least-significant→

Big-Endian bit numbering for BBC/BBS instructions:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

←most-significant   least-significant→

Figure 2–4. Big and Little Bit Numbering for BBC/BBS Instructions

Little-Endian byte addresses, 128-bit processor interface:

127 (←most-significant) (least-significant→) 0

word 0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
word 1	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
word 2															...	32

Big-Endian byte addresses, 128-bit processor interface:

127 (←most-significant) (least-significant→) 0

word 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
word 1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
word 2	32	...														

Little-Endian byte addresses, 64-bit processor interface:

63 (←most-significant) (least-significant→) 0

word 0	7	6	5	4	3	2	1	0
word 1	15	14	13	12	11	10	9	8
word 2							...	16

Big-Endian byte addresses, 64-bit processor interface:

63 (←most-significant) (least-significant→) 0

word 0	0	1	2	3	4	5	6	7
word 1	8	9	10	11	12	13	14	15
word 2	16	...						

Little-Endian byte addresses, 32-bit processor interface:

31 0

word 0	3	2	1	0
word 1	7	6	5	4
word 2			...	8

Big-Endian byte addresses, 32-bit processor interface:

31 0

word 0	0	1	2	3
word 1	4	5	6	7
word 2	8	...		

Figure 2–5. Big and Little Endian Byte Ordering

2.2 Expressions

Table 2–4 defines notational forms used in expressions that describe the operation of instructions. In the table, v is an n-bit quantity, u is an m-bit quantity, and t is a 1-bit quantity.

Table 2–4. Instruction-Description Expressions

Expression Notation[1]	Definition
vx	Bit x of v. The result is 1 bit.
vx..y	Bits from position x to y of v. The result is x-y+1 bits.
vy	The value v replicated y times. The result is n×y bits.
array[i]	Reference to element i of array.
u || v	The catenation of bit strings u and v. The result is m+n bits.
not v	Bitwise logical complement of v. The result is n bits.
u and v	Bitwise logical and of u and v. u and v must be the same width. The result is n bits.
u or v	Bitwise logical or of u and v. u and v must be the same width. The result is n bits.
u xor v	Bitwise logical exclusive or of u and v. u and v must be the same width. The result is n bits.
u = v	Test for exact equality of u and v. u and v must be the same width. The result is 1 bit.
u ≠ v	Test for inequality of u and v. u and v must be the same width. The result is 1 bit.
u < v	Two's complement less-than test on u and v. u and v must be the same width. The result is 1 bit.
u ≤ v	Two's complement less-than or equal-to test on u and v. u and v must be the same width. The result is 1 bit.
u > v	Two's complement greater-than test on u and v. u and v must be the same width. The result is 1 bit.
u ≥ v	Two's complement greater-than or equal-to test on u and v. u and v must be the same width. The result is 1 bit.
u + v	Two's complement addition of u and v. u and v must be the same width. The result is n bits.
u - v	Two's complement subtraction of u and v. u and v must be the same width. The result is n bits.
u x v	Low-order product of two's complement multiplication of u and v. u and v must be the same width. The result is n bits.

[1] t is a 1-bit quantity, u is a m-bit quantity, v is an n-bit quantity. Constants are written either as decimal numbers, in which case the width is determined from context, or in binary.

Table 2–4. Instruction-Description Expressions (continued)

Expression Notation[1]	Definition
u quo v	Quotient of two’s complement division of u by v. u and v must be the same width. The result is n bits.
u rem v	Remainder of two’s complement division of u by v. u and v must be the same width. The result is n bits.
if t then u else v	Conditional expression. The value is u if t = 1. The value is v if t = 0.
$u +_s v$	IEEE754 single-precision floating-point addition of u and v. u and v must be 32 bits. The result is 32 bits.
$u -_s v$	IEEE754 single-precision floating-point subtraction of u and v. u and v must be 32 bits. The result is 32 bits.
$u X_s v$	IEEE754 single-precision floating-point multiplication of u and v. u and v must be 32 bits. The result is 32 bits.
$u \div_s v$	IEEE754 single-precision floating-point division of u by v. u and v must be 32 bits. The result is 32 bits.
$sqrt_s(u)$	IEEE754 single-precision floating-point square root of u. u must be 32 bits. The result is 32 bits.
pows(u,v)	IEEE754 single-precision floating-point power function where u is raised to the v power. u must be 32 bits. The result is 32 bits.

[1] t is a 1-bit quantity, u is a m-bit quantity, v is an n-bit quantity. Constants are written either as decimal numbers, in which case the width is determined from context, or in binary.

2.3 Unsigned Semantics

In this notation, prepending a zero bit is often used for unsigned semantics. For example, the following notation indicates an unsigned less-than test:

```
(0 || u) < (0 || v)
```

2.4 Case

Processor-state variables (for example, registers) are shown in UPPER CASE.

Temporary variables are shown in lower case. If a particular variable is in italics (variable), it is local in the sense that it has no meaning outside the local instruction flow. If it is plain (variable), it comes from or is used outside of the local instruction flow such as an instruction field or the next PC.

2.5 Statements

Table 2–5 defines notational forms used in statements used to describe the operation of instructions.

Table 2–5. Instruction-Description Statements

Statement Notation	Definition
`v ← expr`	Assignment of `expr` to `v`.

`if t1 then`
`      s1`
`[elseif t2 then`
`      s2]`
`.`
`.`
`.`
`[else`
`      sn]`
`endif`
	Conditional statement. If `t1 = 1` then execute statements `s1`. Otherwise, if `t2 = 1` then execute statements `s2`, etc. Finally if none of the previous tests are true, execute statements `sn`.

`label:`	Define `label` for use as a `goto` target.
`goto label`	Transfer control to `label`.

2.6 Instruction Fields

The fields in Table 2–6 are used in the descriptions of the instructions. Instruction formats and opcodes are described in Chapter 7, "Instruction Formats and Opcodes" on page 569.

Table 2–6. Uses Of Instruction Fields

Field	Definition
`op0`	Major opcode
`op1`	4-bit sub-opcode for 24-bit instructions
`op2`	4-bit sub-opcode for 24-bit instructions
`r`	`AR` target (result), `BR` target (result), 4-bit immediate, 4-bit sub-opcode
`s`	`AR` source, `BR` source, `AR` target
`t`	`AR` target, `BR` target, `AR` source, `BR` source, 4-bit sub-opcode

Table 2–6. Uses Of Instruction Fields (continued)

Field	Definition
n	Register window increment, 2-bit sub-opcode, n||2'b00 is used as a AR target on CALLn/CALLXn
m	2-bit sub-opcode
i	1-bit sub-opcode
z	1-bit sub-opcode
imm4	4-bit immediate
imm6	6-bit immediate (PC-relative offset)
imm7	7-bit immediate (for MOVI.N)
imm8	8-bit immediate
imm12	12-bit immediate
imm16	16-bit immediate
offset	18-bit PC-relative offset
ai4const	4-bit immediate, if 0 interpreted as -1, else sign-extended
b4const	4-bit encoded constant value
bbi	5-bit selector for Booleans in registers
sa	4- or 5-bit shift amount
sr	8-bit special register selector
x	1-bit MAC16 data register selector (m0 or m1 only)
y	1-bit MAC16 data register selector (m2 or m3 only)
w	2-bit MAC16 data register selector (m0, m1, m2, or m3)

3. Core Architecture

The Xtensa Core Architecture provides a baseline set of instructions available in every Xtensa implementation. Having such a baseline eases the implementation of core software such as operating system ports and a compiler. This chapter describes that Core Architecture.

3.1 Overview of the Core Architecture

The Xtensa Instruction Set is the product of extensive research into the right balance of features to best address the needs of the embedded processor market. It borrows the best features of other architectures as well as bringing new ISA innovations of its own. While the Xtensa ISA derives most of its features from RISC, it has targeted areas in which older CISC architectures have been strongest, such as compact code.

The Xtensa core ISA is implemented as a set of 24-bit instructions that perform 32-bit operations. The instruction width was chosen primarily with code-size economy in mind. The instructions themselves were selected for their utility in a wide range of embedded applications. The core ISA has many powerful features, such as compound operation instructions, that enhance its fit to embedded applications, but it avoids features that would benefit some applications at the expense of cost or power on others (for example, features that require extra register-file ports). Such features can be implemented in the Xtensa architecture using options and coprocessors specifically targeted at a particular application area.

The Xtensa ISA is organized as a core set of instructions with various optional packages that extend the functionality for specific application areas. This allows the designer to include only the required functionality in the processor core, maximizing the efficiency of the solution. The core ISA provides the functionality required for general control applications, and excels at decision-making and bit and byte manipulation. The core also provides a target for third-party software, and for this reason deletions from the core are not supported. Conversely, numeric computing applications such as digital signal processing are best done with optional ISA packages appropriate for specific application areas, such as the MAC16 Option for integer filters, or the Floating-Point Coprocessor Option for high-end audio processing.

3.2 Processor-Configuration Parameters

Table 3–7 lists the processor-configuration parameters that are required in the core architecture. Additional processor-configuration parameters are listed with each option described in Chapter 4, "Architectural Options" on page 47.

Table 3–7. Core Processor-Configuration Parameters

Parameter	Description	Valid Values
`msbFirst`	Byte order	0 or 1 0 → Little-endian (least significant bit first) 1 → Big-endian (most significant bit first)

3.3 Registers

Table 3–8 lists the core-architecture registers. Each register is described in the sections that follow. Additional registers are added with many of the options described in Chapter 4. The complete set of registers that are predefined in the architecture, including all registers used by the architectural options, is listed in Table 5–127 on page 205.

Table 3–8. Core-Architecture Set

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
`AR`	16[2]	32	Address registers (general registers)	R/W	—
`PC`	1	32	Program counter	R/W	—
`SAR`	1	6	Shift-amount register	R/W	3

[1] Registers with a Special Register assignment are read and/or written with the `RSR`, `WSR`, and `XSR` instructions. See Table 5–127 on page 205. A dash (—) means that the register is not a Special Register.

[2] See "Windowed Register Option" on page 180.

3.3.1 General (AR) Registers

Each instruction contains up to three 4-bit general-register specifiers, each of which can select one of 16 32-bit registers. These general registers are named address registers (`AR`) to distinguish them from coprocessor registers, which in many systems might serve as “data” registers. However, the `AR` registers are not restricted to holding addresses; they can also hold data.

If the Windowed Register Option is configured, the address register file is extended and a mapping from virtual to physical registers is used.

The contents of the address register file are undefined after reset.

3.3.2 Shifts and the Shift Amount Register (SAR)

The ISA provides conventional immediate shifts (logical left, logical right, and arithmetic right), but it does not provide single-instruction shifts in which the shift amount is a register operand. Taking the shift amount from a general register can create a critical timing path. Also, simple shifts do not extend efficiently to larger widths. Funnel shifts (where two data values are catenated on input to the shifter) solve this problem, but require too many operands. The ISA solves both problems by providing a funnel shift in which the shift amount is taken from the SAR register. Variable shifts are synthesized by the compiler using an instruction to compute SAR from the shift amount in a general register, followed by a funnel shift.

Another advantage is that a unidirectional funnel shifter can be manipulated to provide either right or left shifts based on the order of the source operands and transformation of the shift amount. The ISA facilitates implementations that exploit this to reduce the logic required by the shifter.

Funnel shifts are also useful for working with the 40-bit accumulator values created by the MAC16 Option.

To facilitate unsigned bit-field extraction, the EXTUI instructions take a 4-bit mask field that specifies the number of bits to mask the result of the shift. The 4-bit field specifies masks of one to 16 ones. The SRLI instruction provides shifting without a mask.

The legal range of values for SAR is zero to 32, not zero to 31, so SAR is defined as six bits. The use of SRC, SRA, SLL, or SRL when SAR > 32 is undefined.

SAR is undefined after processor reset.

The funnel shifter can also be used efficiently for byte alignment of unaligned memory data. To load four bytes from an arbitrary byte boundary (in a processor that does *not* have the Unaligned Exception Option), use the following code:

```
l32i      a4,a3,0
l32i      a5,a3,4
ssa8l     a3
src       a4,a5,a4
```

An unaligned block copy can be done (in a processor that does *not* have the Unaligned Exception Option) with the following code for little-endian and small changes for big-endian:

```
     l32i      a6,a3,0
     ssa8l     a3
     loopnez   a4,endloop
loop:
     l32i      a7,a3,4
```

```
        src         a8,a7,a6
        s32i        a8,a2,0
        l32i        a6,a3,8
        src         a8,a6,a7
        s32i        a8,a2,4
        addi        a2,a2,8
        addi        a3,a3,8
    endloop:
```

The overhead, compared to an aligned copy, is only one SRC per L32I.

3.3.3 Reading and Writing the Special Registers

The SAR register is part of the Non-Privileged Special Register set in the Xtensa ISA (the other registers in this set are associated with the architectural options). The contents of the special register in the Core Architecture can be read to an AR register with the read special register (RSR.SAR) instruction or written from an AR register with the write special register (WSR.SAR) instruction as shown in Table 3–9. The exchange special register (XSR.SAR) instruction accomplishes the combined action of the read and write instructions.

Table 3–9. Reading and Writing Special Registers

Register Name	Special Register Number	RSR .SAR Instruction	WSR .SAR Instruction
SAR	3	$AR[t] \leftarrow 0^{26} \| SAR$	$SAR \leftarrow AR[t]_{5..0}$

3.4 Data Formats and Alignment

The Core Architecture supports byte, 2-byte, and 4-byte data formats. Two additional data formats are used in architectural options — a 32-bit single-precision format for the Floating-Point Coprocessor Option, and a 40-bit accumulator value for the MAC16 Option. The MAC16 format is not a memory-operand format, but rather a temporary format held in a special 40-bit accumulator register during MAC16 execution; the result can be moved to two 32-bit registers for further operation or storage.

Table 3–10 summarizes the width and alignment of each data type. The processor uses byte addressing for all data types stored in memory (that is, all except the MAC16 accumulator). Byte order can be specified as either big-endian or little-endian. In big-endian byte order, byte 0 is the most-significant (left-most) byte. In little-endian byte order, byte 0 is the least-significant (right-most) byte. When specifying a byte order, both the *byte order* and the *bit order* are specified: the two orderings always have the same most-significant and least-significant ends.

Table 3–10. Operand Formats and Alignment

Operand	Length	Alignment Address in Memory
Byte	8 bit	xxxx
2-byte	16 bits	xxx0
4-byte (word)	32 bits	xx00
IEEE-754 single-precision (Floating-Point Coprocessor Option)	32 bits	xx00
MAC16 accumulator (MAC16 Option)	40 bits	register image only (not in memory)

3.5 Memory

The Xtensa ISA is based on 32-bit virtual and physical memory addresses, which provides a 2^{32} or 4 GB address space for instructions and data.

3.5.1 Memory Addressing

Figure 3–6 shows an example of the processor's interpretation of addresses when configured with caches. The widths of all fields are configurable, and in some cases the width may be zero (in particular, there are always zero ignored bits today). The cache index and cache tag will overlap if the page size is smaller than the size of a single way of the cache and if physical tags are used.

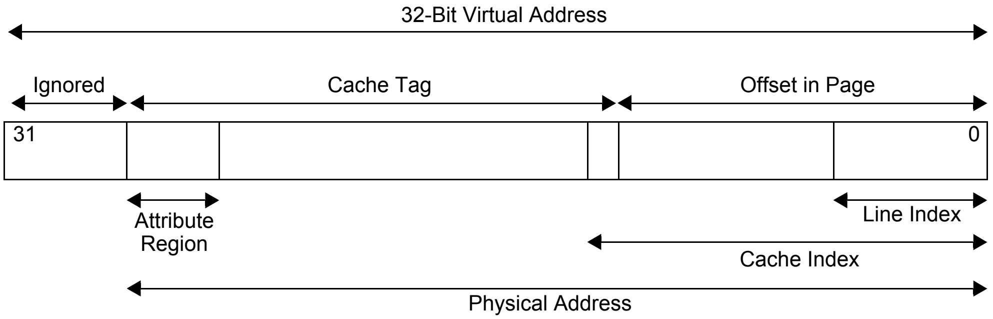


Figure 3–6. Virtual Address Fields

Without the Region Protection Option or the MMU Option, virtual and physical addresses are identical; if physical addresses are configured to be smaller than virtual addresses, virtual addresses are mapped to physical addresses only by truncation (high-order bits are ignored). With the Region Protection Option or the MMU Option, virtual page numbers are translated to physical page numbers.

Without the Region Protection Option or the MMU Option, the formal definition of virtual to physical translation is as follows (note that the `ring` parameter is ignored):

```
function ftranslate(vAddr, ring)-- fetch translate
     b ← $vAddr_{(VABITS-1)..(VABITS-3)}$
     cacheattr ← $CACHEATTR_{(b\|2'b11)..(b\|2'b00)}$
     attributes ← fcadecode(cacheattr)
     cause ← invalid(attributes) then InstructionFetchErrorCause else 0
     ftranslate ← ($vAddr_{PABITS-1..0}$, attributes, cause)
endfunction ftranslate

function ltranslate(vAddr, ring)-- load translate
     b ← $vAddr_{(VABITS-1)..(VABITS-3)}$
     cacheattr ← $CACHEATTR_{(b\|2'b11)..(b\|2'b00)}$
     attributes ← lcadecode(cacheattr)
     cause ← invalid(attributes) then LoadStoreErrorCause else 0
     ltranslate ← ($vAddr_{PABITS-1..0}$, attributes, cause)
endfunction ltranslate

function stranslate(vAddr, ring)-- store translate
     b ← $vAddr_{(VABITS-1)..(VABITS-3)}$
     cacheattr ← $CACHEATTR_{(b\|2'b11)..(b\|2'b00)}$
     attributes ← scadecode(cacheattr)
     cause ← invalid(attributes) then LoadStoreErrorCause else 0
     stranslate ← ($vAddr_{PABITS-1..0}$, attributes, cause)
endfunction stranslate
```

Translation with the MMU Option is described in Section 4.6.5.

The core ISA supports both little-endian (PC compatible) and big-endian (Internet compatible) address models as a configuration parameter. In this manual:

- `msbFirst` = 1 is big-endian.
- `msbFirst` = 0 is little-endian.

3.5.2 Addressing Modes

The core instruction set implements the register + immediate addressing mode. The core ISA does not implement auto-incrementing stores or indexed loads. However, such addressing modes are possible for coprocessors. For example, the Floating-Point Coprocessor Option implements indexed as well as immediate addressing modes.

3.5.3 Program Counter

The 32-bit program counter (PC) holds a byte address and can address 4 GB of virtual memory for instructions. However, when the Windowed Register Option is configured, the register-window call instructions only store the low 30 bits of the return address. Register-window return instructions leave the two most-significant bits of the PC unchanged. Therefore, subroutines called using register window instructions must be placed in the same 1 GB address region as the call.

3.5.4 Instruction Fetch

This section describes the execution loop of the processor using the notation of Chapter 2. The individual instruction actions are represented by the `Inst()` statement, and are detailed in subsequent sections. Two versions of this code are supported; one for little-endian (`msbFirst` = 0) and one for big-endian (`msbFirst` = 1). This definition is in terms of a hypothetical aligned 64-bit fetch, and should not be confused with the fetch algorithms used by specific Xtensa ISA implementations. Aligned 32-bit fetch and unaligned fetch are other possible implementations, which would produce logically equivalent results, but with different timings. Also, actual implementations would be expected to access memory only once for each fetch unit, not once per instruction as in the definition in Section 3.5.4.1 and Section 3.5.4.2.

The processor may speculatively fetch instructions following the address in the program counter. To facilitate this and to allow flexibility in the implementation, software must not position instructions within the last 64 bytes before a boundary where protection or cache attributes change. This exclusion does not apply if one of the two protections or attributes is invalid. Instructions may be placed within 64 bytes before a transition from valid to invalid or from invalid to valid — but not before any other transition. In addition, if the Windowed Register Option is implemented, software must not position instructions within the last 16 bytes of a 2^{30} (1 GB) boundary, to allow flexibility in the implementation of the register-window call and return instructions. The operation of the processor in these exclusion regions is not defined.

3.5.4.1 Little-Endian Fetch Semantics

Little-endian instruction fetch is defined as follows for a 64-bit fetch width (other fetch sizes are similar):

```
checkInterrupts()           -- see "Checking for Interrupts" on page 109
vAddr0 ← $PC_{31..3}$||3'b000      -- this example is 64-bit fetch
(pAddr0, attributes, cause) ← ftranslate(vAddr0, CRING)
if invalid(attributes) then
     EXCVADDR ← vAddr0
     Exception (cause)
     goto abortInstruction
endif
(mem0, error) ← ReadInstMemory(pAddr0, attributes, 8'b11111111)
                                        -- get start of instruction
if error then
     EXCVADDR ← vAddr0
     Exception (InstructionFetchErrorCause)
     goto abortInstruction
endif
$b$ ← $0\|\mathrm{PC}_{2..0}$
if $b_2$ = 0 or $b_1$ = 0 or ($b_0$ = 0 and $mem0_{(b\|3'b011)}$ = 1) then
     -- instruction contained within a single fetch (64 bits in this example)
     inst ← $(\mathrm{undefined}^{64}\|mem0)_{((b+2)\|3'b111)..(b\|3'b000)}$
else
     -- instruction crosses a fetch boundary (64 bits in this example)
     vAddr1 ← vaddr0 + 32'd8
     (pAddr1, attributes, cause) ← ftranslate(vAddr1, CRING)
     if invalid(attributes) then
          EXCVADDR ← vAddr1
          Exception (cause)
          goto abortInstruction
     endif
     (mem1, error) ← ReadInstMemory(pAddr1,
                                    attributes, 8'b11111111)
     if error then
          EXCVADDR ← vAddr1
          Exception (InstructionFetchErrorCause)
          goto abortInstruction
     endif
     inst ← $(mem1\|mem0)_{((b+2)\|3'b111)..(b\|3'b000)}$
endif
-- now have a 24-bit instruction (8 bits undefined if 16-bit), break it into fields
op0 ← $inst_{3..0}$
t ← $inst_{7..4}$
s ← $inst_{11..8}$
r ← $inst_{15..12}$
op1 ← $inst_{19..16}$
op2 ← $inst_{23..20}$
imm8 ← $inst_{23..16}$
imm12 ← $inst_{23..12}$
imm16 ← $inst_{23..8}$
offset ← $inst_{23..6}$
n ← $inst_{5..4}$
m ← $inst_{7..6}$
-- compute nextPC (may be overridden by branches, etc.)
nextPC ← PC + ($0^{30}$ || (if op03 then 2'b10 else 2'b11))
if LCOUNT ≠ $0^{32}$ and CLOOPENABLE and nextPC = LEND then
     LCOUNT ← LCOUNT − 1
     nextPC ← LBEG
```

```
        endif
        -- execute instruction
        Inst()
        checkIcount ()
    abortInstruction:
        PC ← nextPC
```

3.5.4.2 Big-Endian Fetch Semantics

Big-endian instruction fetch is defined as follows for a 64-bit fetch width (other fetch sizes are similar):

```
        checkInterrupts()            -- see "Checking for Interrupts" on page 109
        vAddr0 ← $PC_{31..3}$||3'b000    -- this example is 64-bit fetch
        (pAddr0, attributes, cause) ← ftranslate(vAddr0, CRING)
        if invalid(attributes) then
            EXCVADDR ← vAddr0
            Exception (cause)
            goto abortInstruction
        endif
        (mem0, error) ← ReadInstMemory(pAddr0, attributes, 8'b11111111)
                                     -- get start of instruction
        if error then
            EXCVADDR ← vAddr0
            Exception (InstructionFetchErrorCause)
            goto abortInstruction
        endif
        b ← 0||$PC_{2..0}$
        p0 ← b xor $1^4$
        p2 ← (b + 2) xor $1^4$
        if $b_2$ = 0 or $b_1$ = 0 or ($b_0$ = 0 and $(mem0||undefined^{64})_{(p0||3'b111)}$ = 1)
    then
            -- instruction contained within a single fetch (64 bits in this example)
            inst ← $(mem0||undefined^{64})_{(p0||3'b111)..(p2||3'b000)}$
        else
            -- instruction crosses a fetch boundary (64 bits in this example)
            vAddr1 ← vaddr0 + 32'd8
            (pAddr1, attributes, cause) ← ftranslate(vAddr1, CRING)
            if invalid(attributes) then
                EXCVADDR ← vAddr1
                Exception (cause)
                goto abortInstruction
            endif
            (mem1, error) ← ReadInstMemory(pAddr1,
                                  attributes, 8'b11111111)
            if error then
                EXCVADDR ← vAddr1
                Exception (InstructionFetchErrorCause)
            goto abortInstruction
        endif
        $inst \leftarrow (mem0 \| mem1)_{(p0\|3'b111)..(p2\|3'b000)}$
    endif
    -- now have a 24-bit instruction (8 bits undefined if 16-bit), break it into fields
    $op0 \leftarrow inst_{23..20}$
    $t \leftarrow inst_{19..16}$
    $s \leftarrow inst_{15..12}$
    $r \leftarrow inst_{11..8}$
    $op1 \leftarrow inst_{7..4}$
    $op2 \leftarrow inst_{3..0}$
    $imm8 \leftarrow inst_{7..0}$
    $imm12 \leftarrow inst_{11..0}$
    $imm16 \leftarrow inst_{15..0}$
    $offset \leftarrow inst_{17..0}$
    $n \leftarrow inst_{19..18}$
    $m \leftarrow inst_{17..16}$
    -- compute nextPC (may be overridden by branches, etc.)
    $nextPC \leftarrow PC + (0^{30} \| (\text{if } op0_3 \text{ then } 2'b10 \text{ else } 3'b11))$
    $\text{if } LCOUNT \neq 0^{32} \text{ and } CLOOPENABLE \text{ and } nextPC = LEND \text{ then}$
        $LCOUNT \leftarrow LCOUNT - 1$
        $nextPC \leftarrow LBEG$
    endif
    -- execute instruction
    Inst()
    checkIcount ()
abortInstruction:
    $PC \leftarrow nextPC$
```

3.6 Reset

When the processor emerges from the reset state, it initializes many registers. The ISA guarantees the values of some states after reset but leaves many others undefined. Actual Xtensa processor implementations will often define the values of state left undefined by the ISA. Chapter 5, "Processor State" on page 205 contains information about each state value, including the value to which it is reset.

3.7 Exceptions and Interrupts

The core ISA does not include support for exceptions or interrupts. These are architectural options are described in Section 4.4. Software running on a processor that is configured without an Exception Option should be well tested, as such a processor will do something unexpected if it encounters a software error.

3.8 Instruction Summary

Table 3–11 summarizes the core instructions included in all versions of the Xtensa architecture. The remainder of this section gives an overview of the core instructions.

Table 3–11. Core Instruction Summary

Instruction Category	Instructions[1]	Reference
Load	L8UI, L16SI, L16UI, L32I, L32R	"Load Instructions" on page 33
Store	S8I, S16I, S32I	"Store Instructions" on page 36
Memory ordering	MEMW, EXTW	"Memory Access Ordering" on page 39
Jump, Call	CALL0, CALLX0, RET J, JX	"Jump and Call Instructions" on page 40
Conditional branch	BALL, BNALL, BANY, BNONE BBC, BBCI, BBS, BBSI BEQ, BEQI, BEQZ BNE, BNEI, BNEZ BGE, BGEI, BGEU, BGEUI, BGEZ BLT, BLTI, BLTU, BLTUI, BLTZ	"Conditional Branch Instructions" on page 40
Move	MOVI, MOVEQZ, MOVGEZ, MOVLTZ, MOVNEZ	"Move Instructions" on page 42
Arithmetic	ADDI, ADDMI, ADD, ADDX2, ADDX4, ADDX8, SUB, SUBX2, SUBX4, SUBX8, NEG, ABS	"Arithmetic Instructions" on page 43
Bitwise logical	AND, OR, XOR	"Bitwise Logical Instructions" on page 44
Shift	EXTUI, SRLI, SRAI, SLLI SRC, SLL, SRL, SRA SSL, SSR, SSAI, SSA8B, SSA8L	"Shift Instructions" on page 44
Processor control	RSR, WSR, XSR, RUR, WUR, ISYNC, RSYNC, ESYNC, DSYNC, NOP	"Processor Control Instructions" on page 45

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

3.8.1 Load Instructions

Load instructions form a virtual address by adding a base register and an 8-bit unsigned offset. This virtual address is translated to a physical address if necessary. The physical address is then used to access the memory system (often through a cache). The memory system returns a data item (either 32, 64, or 128 bits, depending on the configuration). The load instructions then extract the referenced data from that memory item and either zero-extend or sign-extend the result to be written into a register. Unless the

Unaligned Exception Option is enabled, the processor does not handle misaligned data or trap when a misaligned address is used; instead it simply loads the aligned data item containing the computed virtual address. This allows the funnel shifter to be used with a pair of loads to reference data on any byte address.

Only the loads `L32I`, `L32I.N`, and `L32R` can access InstRAM and InstROM locations.

Table 3–12 shows the loads in the Core Architecture.

Table 3–12. Load Instructions

Instruction	Format	Definition
`L8UI`	RRI8	8-bit unsigned load (8-bit offset)
`L16SI`	RRI8	16-bit signed load (8-bit shifted offset)
`L16UI`	RRI8	16-bit unsigned load (8-bit shifted offset)
`L32I`	RRI8	32-bit load (8-bit shifted offset)
`L32R`	RI16	32-bit load PC-relative (16-bit negative word offset)

Because the operation of caches is implementation-specific, this manual does not provide a formal specification of cache access.

The following routines define the load instructions:

```
function ReadMemory (pAddr, attributes, bytemask)
     ReadMemory ← (Memory[pAddr], 0)              -- for now, no cache
endfunction ReadMemory

function Load8 (vAddr)
     (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
     if invalid(attributes) then
          EXCVADDR ← vAddr
          Exception (cause)
          goto abortInstruction
     endif
     p ← $pAddr_{2..0}$ xor $msbFirst^{3}$
     (mem64, error) ← ReadMemory($pAddr_{31..3}$, attributes, $0^{7-p}||1||0^{p}$)
     mem8 ← $mem64_{(p||3'b111)..(p||3'b000)}$
     Load8 ← (mem8, error)
endfunction Load8

function Load16 (vAddr)
     if UnalignedExceptionOption & $Vaddr_{0}$ ≠ 1’b0 then
          EXCVADDR ← vAddr
          Exception (LoadStoreAlignmentCause)
          goto abortInstruction
```

```
    endif
    (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
        goto abortInstruction
    endif
    p ← $pAddr_{2..1}$ xor $msbFirst^2$
    (mem64, error) ← ReadMemory($pAddr_{31..3}$, attributes,
                          $(2'b00)^{3-p} \| 2'b11 \| (2'b00)^p$)
    mem16 ← $mem64_{(p\|4'b1111)..(p\|4'b0000)}$
    Load16 ← (mem16, error)
endfunction Load16

function Load32 (vAddr)
    if UnalignedExceptionOption & $Vaddr_{1..0} \neq$ 2’b00 then
        EXCVADDR ← vAddr
        Exception (LoadStoreAlignmentCause)
        goto abortInstruction
    endif
    (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
        goto abortInstruction
    endif
    p ← pAddr2 xor msbFirst
    (mem64, error) ← ReadMemory(pAddr31..3, attributes,
                          $(4'b0000)^{1-p} \| 4'b1111 \| (4'b0000)^p$)
    mem32 ← $mem64_{(p\|5'b11111)..(p\|5'b00000)}$
    Load32 ← (mem32, error)
endfunction Load32

function Load32Ring (vAddr, ring)
    if UnalignedExceptionOption & $Vaddr_{1..0} \neq$ 2’b00 then
        EXCVADDR ← vAddr
        Exception (LoadStoreAlignmentCause)
        goto abortInstruction
    endif
    (pAddr, attributes, cause) ← ltranslate(vAddr, ring)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
        goto abortInstruction
    endif
    p ← $pAddr_2$ xor msbFirst
    (mem64, error) ← ReadMemory($pAddr_{31..3}$, attributes,
                          $(4'b0000)^{1-p} \| 4'b1111 \| (4'b0000)^p$)
    mem32 ← $mem64_{(p\|5'b11111)..(p\|5'b00000)}$
```

```
        Load32 ← (mem32, error)
    endfunction Load32Ring

    function Load64 (vAddr)
        if UnalignedExceptionOption & $Vaddr_{2..0} \ne$ 3’b000 then
            EXCVADDR ← vAddr
            Exception (LoadStoreAlignmentCause)
            goto abortInstruction
        endif
        (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
        if invalid(attributes) then
            EXCVADDR ← vAddr
            Exception (cause)
            goto abortInstruction
        endif
        Load64 ← ReadMemory($pAddr_{31..3}$, attributes, 8'b11111111)
    endfunction Load64
```

3.8.2 Store Instructions

Store instructions are similar to load instructions in address formation. Store memory errors are not synchronous exceptions; it is expected that the memory system will use an interrupt to indicate an error on a store.

Only the stores S32I and S32I.N can access InstRAM.

Table 3–13 shows the loads in the Core Architecture.

Table 3–13. Store Instructions

Instruction	Format	Definition
S8I	RRI8	8-bit store (8-bit offset)
S16I	RRI8	16-bit store (8-bit shifted offset)
S32I	RRI8	32-bit store (8-bit shifted offset)

The following routines define the store instructions:

```
    procedure WriteMemory (pAddr, attributes, bytemask, data64)
        -- for now, no cache
        if $bytemask_0$ then
            $Memory[pAddr]_{7..0}$ ← $data64_{7..0}$
        endif
        if $bytemask_1$ then
            $Memory[pAddr]_{15..8}$ ← $data64_{15..8}$
        endif
        if $bytemask_2$ then
```

```
            $\text{Memory[pAddr]}_{23..16} \leftarrow \text{data64}_{23..16}$
        endif
        if $\text{bytemask}_3$ then
            $\text{Memory[pAddr]}_{31..24} \leftarrow \text{data64}_{31..24}$
        endif
        if $\text{bytemask}_4$ then
            $\text{Memory[pAddr]}_{39..32} \leftarrow \text{data64}_{39..32}$
        endif
        if $\text{bytemask}_5$ then
            $\text{Memory[pAddr]}_{47..40} \leftarrow \text{data64}_{47..40}$
        endif
        if $\text{bytemask}_6$ then
            $\text{Memory[pAddr]}_{55..48} \leftarrow \text{data64}_{55..48}$
        endif
        if $\text{bytemask}_7$ then
            $\text{Memory[pAddr]}_{63..56} \leftarrow \text{data64}_{63..56}$
        endif
    endprocedure WriteMemory

    procedure Store8 (vAddr, data8)
        (pAddr, attributes, cause) ← stranslate(vAddr, CRING)
        if invalid(attributes) then
            EXCVADDR ← vAddr
            Exception (cause)
            goto abortInstruction
        endif
        $p \leftarrow pAddr_{2..0}$ xor $\text{msbFirst}^3$
        WriteMemory($pAddr_{31..3}$, attributes, $0^{7-p}\|1\|0^p$,
                    $\text{undefined}^{(7-p)\|3\text{'b000}}\|\text{data8}\|\text{undefined}^{p\|3\text{'b000}}$)
    endprocedure Store8

    procedure Store16 (vAddr, data16)
        if UnalignedExceptionOption & $\text{Vaddr}_0 \neq 1\text{'b0}$ then
            EXCVADDR ← vAddr
            Exception (LoadStoreAlignmentCause)
            goto abortInstruction
        endif
        (pAddr, attributes, cause) ← stranslate(vAddr, CRING)
        if invalid(attributes) then
            EXCVADDR ← vAddr
            Exception (cause)
            goto abortInstruction
        endif
        $p \leftarrow pAddr_{2..1}$ xor $\text{msbFirst}^2$
        WriteMemory($pAddr_{31..3}$, attributes, $(2\text{'b00})^{3-p}\|2\text{'b11}\|(2\text{'b00})^p$,
                    $\text{undefined}^{(3-p)\|4\text{'b0000}}\|\text{data16}\|\text{undefined}^{p\|4\text{'b0000}}$)
    endprocedure Store16

    procedure Store32 (vAddr, data32)
```

```
        if UnalignedExceptionOption & Vaddr$_{1..0}$ ≠ 2’b00 then
            EXCVADDR ← vAddr
            Exception (LoadStoreAlignmentCause)
            goto abortInstruction
        endif
        (pAddr, attributes, cause) ← stranslate(vAddr, CRING)
        if invalid(attributes) then
            EXCVADDR ← vAddr
            Exception (cause)
            goto abortInstruction
        endif
        p ← pAddr$_2$ xor msbFirst
        WriteMemory(pAddr$_{31..3}$, attributes, (4'b0000)$^{1-p}$||4'b1111||(4'b0000)$^{p}$,
                    undefined$^{(1-p)||5'b00000}$||data32||undefined$^{p||5'b00000)}$
    endprocedure Store32

    procedure Store32Ring (vAddr, data32, ring)
        if UnalignedExceptionOption & Vaddr$_{1..0}$ ≠ 2’b00 then
            EXCVADDR ← vAddr
            Exception (LoadStoreAlignmentCause)
            goto abortInstruction
        endif
        (pAddr, attributes, cause) ← stranslate(vAddr, ring)
        if invalid(attributes) then
            EXCVADDR ← vAddr
            Exception (cause)
            goto abortInstruction
        endif
        p ← pAddr$_2$ xor msbFirst
        WriteMemory(pAddr$_{31..3}$, attributes, (4'b0000)$^{1-p}$||4'b1111||(4'b0000)$^{p}$,
                    undefined$^{(1-p)||5'b00000}$||data32||undefined$^{p||5'b00000)}$
    endprocedure Store32Ring

    procedure Store64 (vAddr, data64)
        if UnalignedExceptionOption & Vaddr$_{2..0}$ ≠ 3’b000 then
            EXCVADDR ← vAddr
            Exception (LoadStoreAlignmentCause)
            goto abortInstruction
        endif
        (pAddr, attributes, cause) ← stranslate(vAddr, CRING)
        if invalid(attributes) then
            EXCVADDR ← vAddr
            Exception (cause)
            goto abortInstruction
        endif
        WriteMemory(pAddr$_{31..3}$, attributes, 8'b11111111, data64)
    endprocedure Store64
```

3.8.3 Memory Access Ordering

Xtensa implementations can perform ordinary load and store operations in any order, as long as loads return the last (as defined by program execution order) values stored to each byte of the load address for a single processor and a simple memory. This flexibility is appropriate because most memory accesses require only these semantics and some implementations may be able to execute programs significantly faster by exploiting non-program order memory access. The Xtensa ISA only requires that implementations follow a simplified version of the Release Consistency model[1] of memory access ordering, although many implement stricter orderings for simplicity. For more on the Xtensa memory order semantics, see "Multiprocessor Synchronization Option" on page 74.

However, some load and store instructions are executed not just to read and write storage, but to cause some side effects on some other part of the system (for example, another processor or an I/O device). In C and C++, such variables must be declared `volatile`. Loads and stores to such locations must be executed in program order. The Xtensa ISA therefore provides an instruction that can be used to give program ordering of load and store memory accesses.

The `MEMW` instruction causes all memory and cache accesses (loads, stores, acquires, releases, prefetches, and cache operations, but *not* instruction fetches) before itself in program order to access memory before all memory and cache accesses (but *not* instruction fetches) after. At least one `MEMW` should be executed in between every load or store to a `volatile` variable. The Multiprocessor Synchronization Option provides some additional instructions that also affect memory ordering in a more focused fashion. `MEMW` has broader applications than these other instructions (for example, when reading and writing device registers), but it also may affect performance more than the synchronization instructions.

The `EXTW` instruction is similar to `MEMW`, but it separates all external effects of instructions before the `EXTW` in program order from all external effects of instructions after the `EXTW` in program order. `EXTW` is a superset of `MEMW`, and includes memory accesses in what it orders.

Table 3–14 shows the memory ordering instructions in the Core Architecture.

Table 3–14. Memory Order Instructions

Instruction	Format	Definition
`MEMW`	RRR	Order memory accesses before with memory access after
`EXTW`	RRR	Order all external effects before with all external effects after

[1] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy, "*Memory consistency and event ordering in scalable shared-memory multiprocessors*," Proceedings of the 17th Annual International Symposium on Computer Architecture, pages 15-26, May 1990.

3.8.4 Jump and Call Instructions

The unconditional branch instruction, J, has a longer range (PC-relative) than conditional branches. Calls have a slightly longer range because they target 32-bit aligned addresses. In addition, jump and call indirect instructions provide support for case dispatch, function variables, and dynamic linking.

Table 3–15 shows the jump and call instructions.

Table 3–15. Jump and Call Instructions

Instruction	Format	Definition
CALL0	CALL	Call subroutine, PC-relative
CALLX0	CALLX	Call subroutine, address in register
J	CALL	Unconditional jump, PC-relative
JX	CALLX	Unconditional jump, address in register
RET	CALLX	Subroutine return—jump to return address. Used to return from a routine called by CALL0/CALLX0.

3.8.5 Conditional Branch Instructions

The branch instructions in Table 3–16 compare a register operand against zero, an immediate, or a second register value and conditional branch based on the result of the comparison. Compound compare and branch instructions improve code density and performance compared to other ISAs. All branches are PC-relative; the immediate field contains the difference between the target PC and the current PC plus four. The use of a PC-relative offset of minus three to zero is illegal and reserved for future use.

Table 3–16. Conditional Branch Instructions

Instruction	Format	Definition
BEQZ	BRI12	Branch if equal to zero
BNEZ	BRI12	Branch if not equal to zero
BGEZ	BRI12	Branch if greater than or equal to zero
BLTZ	BRI12	Branch if less than zero
BEQI	BRI8	Branch if equal immediate[1]
BNEI	BRI8	Branch if not equal immediate[1]
BGEI	BRI8	Branch if greater than or equal immediate[1]
BLTI	BRI8	Branch if less than immediate[1]
BGEUI	BRI8	Branch if greater than or equal unsigned immediate[2]

[1] See Table 3–17 for encoding of signed immediate constants.

[2] See Table 3–18 for encoding of unsigned immediate constants.

Table 3–16. Conditional Branch Instructions (continued)

Instruction	Format	Definition
BLTUI	BRI8	Branch if less than unsigned immediate[2]
BBCI	RRI8	Branch if bit clear immediate
BBSI	RRI8	Branch if bit set immediate
BEQ	RRI8	Branch if equal
BNE	RRI8	Branch if not equal
BGE	RRI8	Branch if greater than or equal
BLT	RRI8	Branch if less than
BGEU	RRI8	Branch if greater than or equal unsigned
BLTU	RRI8	Branch if less than Unsigned
BANY	RRI8	Branch if any of masked bits set
BNONE	RRI8	Branch if none of masked bits set (All Clear)
BALL	RRI8	Branch if all of masked bits set
BNALL	RRI8	Branch if not all of masked bits set
BBC	RRI8	Branch if bit clear
BBS	RRI8	Branch if bit set

1. See Table 3–17 for encoding of signed immediate constants.
2. See Table 3–18 for encoding of unsigned immediate constants.

The encodings for the branch immediate constant (`b4const`) field and the branch unsigned immediate constant (`b4constu`) fields, shown in Table 3–17 and Table 3–18, specify one of the sixteen most frequent compare immediates for each type of constant.

Table 3–17. Branch Immediate (`b4const`) Encodings

Encoding	Decimal Value of Immediate	Hex Value of Immediate
0	-1	32’hFFFFFFFF
1	1	32’h00000001
2	2	32’h00000002
3	3	32’h00000003
4	4	32’h00000004
5	5	32’h00000005
6	6	32’h00000006
7	7	32’h00000007
8	8	32’h00000008
9	10	32’h0000000A
10	12	32’h0000000C

Table 3–17. Branch Immediate (`b4const`) Encodings (continued)

Encoding	Decimal Value of Immediate	Hex Value of Immediate
11	16	32’h00000010
12	32	32’h00000020
13	64	32’h00000040
14	128	32’h00000080
15	256	32’h00000100

Table 3–18. Branch Unsigned Immediate (`b4constu`) Encodings

Encoding	Decimal Value of Immediate	Hex Value of Immediate
0	32768	32’h00008000
1	65536	32’h00010000
2	2	32’h00000002
3	3	32’h00000003
4	4	32’h00000004
5	5	32’h00000005
6	6	32’h00000006
7	7	32’h00000007
8	8	32’h00000008
9	10	32’h0000000A
10	12	32’h0000000C
11	16	32’h00000010
12	32	32’h00000020
13	64	32’h00000040
14	128	32’h00000080
15	256	32’h00000100

3.8.6 Move Instructions

`MOVI` sets a register to a constant encoded in the instruction. The conditional move instructions shown in Table 3–19 are used for branch avoidance.

Table 3–19. Move Instructions

Instruction	Format	Definition
MOVI	RRI8	Load register with 12-bit signed constant
MOVEQZ	RRR	Conditional move if zero
MOVNEZ	RRR	Conditional move if non-zero
MOVLTZ	RRR	Conditional move if less than zero
MOVGEZ	RRR	Conditional move if greater than or equal to zero

3.8.7 Arithmetic Instructions

The arithmetic instructions that Table 3–20 lists include add and subtract with a small shift for address calculations and for synthesizing constant multiplies. The ADDMI instruction is included for extending the range of load and store instructions.

Table 3–20. Arithmetic Instructions

Instruction	Format	Definition
ADD	RRR	Add two registers $AR[r] \leftarrow AR[s] + AR[t]$
ADDX2	RRR	Add register to register shifted by 1 $AR[r] \leftarrow (AR[s]_{30..0} \parallel 0) + AR[t]$
ADDX4	RRR	Add register to register shifted by 2 $AR[r] \leftarrow (AR[s]_{29..0} \parallel 0^{2}) + AR[t]$
ADDX8	RRR	Add register to register shifted by 3 $AR[r] \leftarrow (AR[s]_{28..0} \parallel 0^{3}) + AR[t]$
SUB	RRR	Subtract two registers $AR[r] \leftarrow AR[s] - AR[t]$
SUBX2	RRR	Subtract register from register shifted by 1 $AR[r] \leftarrow (AR[s]_{30..0} \parallel 0) - AR[t]$
SUBX4	RRR	Subtract register from register shifted by 2 $AR[r] \leftarrow (AR[s]_{29..0} \parallel 0^{2}) - AR[t]$
SUBX8	RRR	Subtract register from register shifted by 3 $AR[r] \leftarrow (AR[s]_{28..0} \parallel 0^{3}) - AR[t]$
NEG	RRR	Negate $AR[r] \leftarrow 0 - AR[t]$

Table 3–20. Arithmetic Instructions (continued)

Instruction	Format	Definition
ABS	RRR	Absolute value AR[r] ← if $AR[s]_{31}$ then 0 − AR[s] else AR[s]
ADDI	RRI8	Add signed constant to register AR[t] ← AR[s] + $(imm8_7^{24} \Vert imm8)$
ADDMI	RRI8	Add signed constant shifted by 8 to register AR[t] ← AR[s] + $(imm8_7^{16} \Vert imm8 \Vert 0^8)$

3.8.8 Bitwise Logical Instructions

The bitwise logical instructions in Table 3–21 provide a core set from which other logicals can be synthesized. Immediate forms of these instructions are not provided because the immediate would be only four bits.

Table 3–21. Bitwise Logical Instructions

Instruction	Format	Definition
AND	RRR	Bitwise logical AND AR[r] ← AR[s] and AR[t]
OR	RRR	Bitwise logical OR AR[r] ← AR[s] or AR[t]
XOR	RRR	Bitwise logical exclusive OR AR[r] ← AR[s] xor AR[t]

3.8.9 Shift Instructions

The shift instructions in Table 3–22 provide a rich set of operations while avoiding critical timing paths. See Section 3.3.2 on page 25 for more information.

Table 3–22. Shift Instructions

Instruction	Format	Definition
EXTUI	RRR	Extract unsigned field immediate Shifts right by 0..31 and ANDs with a mask of 1..16 ones The operation of this instruction when the number of mask bits exceeds the number of significant bits remaining after the shift is undefined and reserved for future use.
SLLI	RRR	Shift left logical immediate by 1..31 bit positions (see page 525 for encoding of the immediate value).
SRLI	RRR	Shift right logical immediate by 0..15 bit positions There is no SRLI for shifts ≥ 16; use EXTUI instead.
SRAI	RRR	Shift right arithmetic immediate by 0..31 bit positions

Table 3–22. Shift Instructions (continued)

Instruction	Format	Definition
SRC	RRR	Shift right combined (a funnel shift with shift amount from SAR) The two source registers are catenated, shifted, and the least significant 32 bits returned.
SRA	RRR	Shift right arithmetic (shift amount from SAR)
SLL	RRR	Shift left logical (Funnel shift AR[s] and 0 by shift amount from SAR)
SRL	RRR	Shift right logical (Funnel shift 0 and AR[s] by shift amount from SAR)
SSA8B	RRR	Set shift amount register (SAR) for big-endian byte align The t field must be zero.
SSA8L	RRR	Set shift amount register (SAR) for little-endian byte align
SSR	RRR	Set shift amount register (SAR) for shift right logical This instruction differs from WSR to SAR in that only the five least significant bits of the register are used.
SSL	RRR	Set shift amount register (SAR) for shift left logical
SSAI	RRR	Set shift amount register (SAR) immediate

3.8.10 Processor Control Instructions

Table 3–23 contains processor control instructions. The RSR.*, WSR.*, and XSR.* instructions read, write, and exchange Special Registers for both the Core Architecture and the architectural options, as detailed in Table 5–128 on page 209. They save and restore context, process interrupts and exceptions, and control address translation and attributes. The XSR.* instruction reads and writes both the Special Register, and AR[t]. It combines the RSR.* and WSR.* operations to exchange the Special Register with AR[t]. The XSR.* instruction is not present in T1030 and earlier processors.

The xSYNC instructions synchronize Special Register writes and their uses. See Chapter 5 for more information on how xSYNC instructions are used. These synchronization instructions are separate from the synchronization instructions used for multiprocessors, which are described in Section 4.3.12 on page 74.

On some Xtensa implementations the latency of RSR is greater than one cycle, and so it is advantageous to schedule uses of the RSR result away from the RSR to avoid an interlock.

The point at which WSR.* or XSR.* to most Special Registers affects subsequent instructions is not defined (SAR and ACC are exceptions). In these cases, Table 5–128 on page 209 explains how to ensure the effects are seen by a particular point in the instruction stream (typically involving the use of one of the ISYNC, RSYNC, ESYNC, or DSYNC instructions). A WSR.* or XSR.* followed by a RSR.* of the same register must be separated by an ESYNC instruction to guarantee the value written is read back. A WSR.PS or XSR.PS followed by a RSIL also requires an ESYNC instruction.

Table 3–23. Processor Control Instructions

Instruction	Format	Definition
RSR	RSR	Read Special Register
WSR	RSR	Write Special Register
XSR	RSR	Exchange Special Register (combined RSR and WSR) Not present in T1030 and earlier processors
ISYNC	RRR	Instruction fetch synchronize: Waits for all previously fetched load, store, cache, and special register write instructions that affect instruction fetch to be performed before fetching the next instruction.
RSYNC	RRR	Instruction register synchronize: Waits for all previously fetched WSR and XSR instructions to be performed before interpreting the register fields of the next instruction. This operation is also performed as part of ISYNC.
ESYNC	RRR	Register value synchronize: Waits for all previously fetched WSR and XSR instructions to be performed before the next instruction uses any register values. This operation is also performed as part of ISYNC and RSYNC.
DSYNC	RRR	Load/store synchronize: Waits for all previously fetched WSR and XSR instructions to be performed before interpreting the virtual address of the next load or store instruction. This operation is also performed as part of ISYNC, RSYNC, and ESYNC.
NOP	RRR	No operation

4. Architectural Options

This chapter defines the Xtensa ISA options. Each option adds some associated configuration resources and capabilities. Some options are dependent on the implementation of other options. These interdependencies, if any, are listed as *Prerequisites* at the beginning of the description of each option. The additional parameters required to define the option, the new state and instructions added by the option, and any other new features (such as exceptions) added by the option are listed and the operation of the option is described.

4.1 Overview of Options

Section 4.2 provides a synopsis of the Core Architecture (covered in more detail in Chapter 3) in a format similar to the format used for the options. The Instruction Set options available with an Xtensa processor are listed in five groups below.

"Options for Additional Instructions" on page 53 lists options whose primary function is to add new instructions to the processor's instruction set, including:

- The **Code Density Option** on page 53 adds 16-bit encodings of the most frequently used 24-bit instructions for higher code density.
- The **Loop Option** on page 54 adds a "zero overhead loop," which requires neither the extra instruction for a branch at the end of a loop nor the additional delay slots that would result from the taken branch. A few fixed cycles of overhead mean that each iteration of the loop pays no cost for the loop branch.
- The **Extended** L32R Option on page 56 allows an additional choice in the addressing mode of the `L32R` instruction.
- The **16-bit Integer Multiply Option** on page 57 adds signed and unsigned 16x16 multiplication instructions that produce 32-bit results.
- The **32-bit Integer Multiply Option** on page 58 adds signed and unsigned 32x32 multiplication instructions that produce high and low parts of a 64-bit result.
- The **32-bit Integer Divide Option** on page 59 implements signed and unsigned 32-bit division and remainder instructions.
- The **MAC16 Option** on page 60 adds multiply-accumulate functions that are useful in digital signal processing (DSP).
- The **Miscellaneous Operations Option** on page 62 provides a series of instructions useful for some applications, but which are not necessary for others. By making these optional, the Xtensa architecture allows the designer to choose only those additional instructions that benefit the application.

- The **Coprocessor Option** on page 63 allows the grouping of certain states in the processor and adds an enable bit, which allows for lazy context switching.
- The **Boolean Option** on page 65 adds a set of Boolean registers, which can be set and cleared by user instructions and that can be used as branch conditions.
- The **Floating-Point Coprocessor Option** on page 67 adds a floating-point unit for single precision floating point.
- The **Multiprocessor Synchronization Option** on page 74 adds acquire and release instructions with specific memory ordering relationships to the other Xtensa memory access instructions.
- The **Conditional Store Option** on page 77 adds a compare and swap type atomic operation to the instruction set.

"Options for Interrupts and Exceptions" on page 82 lists options whose primary function is to add and control exceptions and interrupts, including:

- The **Exception Option** on page 82 adds the basic functions needed for the processor to take exceptions.
- The **Relocatable Vector Option** on page 98 adds the ability for the exception vectors to be relocated at run time.
- The **Unaligned Exception Option** on page 99 adds an exception for memory accesses that are not aligned by their own size. They may then be emulated in software.
- The **Interrupt Option** on page 100 builds upon the Exception Option to add a flexible software prioritized interrupt system.
- The **High-Priority Interrupt Option** on page 106 adds a hardware prioritized interrupt system for higher performance.
- The **Timer Interrupt Option** on page 110 adds timers and interrupts, which are caused when the timer expires.

"Options for Local Memory" on page 111 lists options whose primary function is to add different kinds of memory, such as RAMs, ROMs, or caches to the processor, including:

- The **Instruction Cache Option** on page 115 adds an interface for a direct-mapped or set-associative instruction cache.
- The **Instruction Cache Test Option** on page 116 adds instructions to access the instruction cache tag and data.
- The **Instruction Cache Index Lock Option** on page 117 adds per-index locking to the instruction cache.
- The **Data Cache Option** on page 118 adds an interface for a direct-mapped or set-associative data cache.
- The **Data Cache Test Option** on page 121 adds instructions to access the data cache tag.

- The **Data Cache Index Lock Option** on page 122 adds per-index locking to the data cache.
- The **Instruction RAM Option** on page 124 adds an interface for a local instruction memory.
- The **Instruction ROM Option** on page 125 adds an interface for a local instruction Read Only Memory.
- The **Data RAM Option** on page 126 adds an interface for a local data memory.
- The **Data ROM Option** on page 126 adds an interface for a local data read-only memory.
- The **XLMI Option** on page 127 adds an interface with the timing of the local memory interfaces, but with a full enough signal set to support non-memory devices.
- The **Hardware Alignment Option** on page 128 adds the ability for the hardware to handle unaligned accesses to data memory.
- The **Memory ECC/Parity Option** on page 128 provides the ability to add parity or ECC to cache and local memories.

"Options for Memory Protection and Translation" on page 138 lists options whose primary function is to control access to and manage memory, including:

- The **Region Protection Option** on page 150 adds protection on memory in eight segments.
- The **Region Translation Option** on page 156 adds protection on memory in eight segments and allows translations from one segment to another.
- The **MMU Option** on page 158 adds full paging virtual memory management hardware.

"Options for Other Purposes" on page 179 lists options that do not fall conveniently into one of the other groups, including:

- The **Windowed Register Option** on page 180 adds additional physical `AR` registers and a mapping mechanism, which together lead to smaller code size and higher performance.
- The **Processor Interface Option** on page 194 adds a bus interface used by memory accesses, which are to locations other than local memories. It is used for cache misses for cacheable addresses as well as for cache bypass memory accesses.
- The **Miscellaneous Special Registers Option** on page 195 provides one to four scratch registers within the processor readable and writable by `RSR`, `WSR`, and `XSR`, which may be used for application-specific exceptions and interrupt processing tasks.
- The **Thread Pointer Option** on page 196 provides a Special Register that may be used for a thread pointer.

- The **Processor ID Option** on page 196 adds a register that software can use to distinguish which of several processors it is running on.
- The **Debug Option** on page 197 adds instructions-counting and breakpoint exceptions for debugging by software or external hardware.
- The **Trace Port Option** on page 203 architectural features for supporting hardware tracing of the processor.

The functionality of a fairly complete micro-controller is provided by enabling the Code Density Option, the Exception Option, the Interrupt Option, the High-Priority Interrupt Option, the Timer Interrupt Option, the Debug Option, and the Windowed Register Option.

The primary reason to disable the Code Density Option (16-bit instructions) is to provide maximum opcode space for extensions. The primary reason to disable the other options listed above is reduce the processor core area.

The choice of Cache, RAM, or ROM Options for instruction and data depends on the characteristics of the application. RAM is not as flexible as Cache, but it requires slightly less area because tags are not required. RAM may also be desirable when performance predictability is required. ROM is even less flexible than RAM, but avoids the need to load the memory and offers some protection from program errors and tampering.

4.2 Core Architecture

The Core Architecture is not an option, but rather a minimum base of processor state and instructions, which allows system software and compiled code to run on all Xtensa implementations. There are no prerequisites or incompatible options, but the tables normally used to show option additions are used here to give the base set. Table 4–24 through Table 4–26 show Core Architecture processor configurations, processor state, and instructions.

Table 4–24. Core Architecture Processor-Configurations

Parameter	Description	Valid Values
`msbFirst`	Byte order for memory accesses	0 or 1 0 → Little-endian (least significant bit first) 1 → Big-endian (most significant bit first)

Table 4–25. Core Architecture Processor-State

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
AR	16	32	Address register file	R/W	—
PC	1	32	Program counter	—	—
SAR	1	6	Shift amount register	R/W	3

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 3–23 on page 46.

Table 4–26. Core Architecture Instructions

Instruction[1]	Format	Definition
ABS	RRR	Absolute value
ADD	RRR	Add two registers
ADDI	RRI8	Add a register and an 8-bit immediate
ADDMI	RRI8	Add a register and a shifted 8-bit immediate
ADDX2/4/8	RRR	Add two registers with one of them shifted left by one/two/three
AND	RRR	Bitwise AND of two registers
BALL/BANY	RRI8	Branch if all/any bits specified by a mask in one register are set in another register
BBC/BBS	RRI8	Branch if the bit specified by another register is clear/set
BBCI/BBSI	RRI8	Branch if the bit specified by an immediate is clear/set
BEQ	RRI8	Branch if a register equals another register
BEQI	RRI8	Branch if a register equals an encoded constant
BEQZ	BRI12	Branch if a register equals zero
BGE	RRI8	Branch if one register is greater than or equal to a register
BGEI	RRI8	Branch if one register is greater than or equal to an encoded constant
BGEU	RRI8	Branch if one register is greater or equal to a register as unsigned
BGEUI	BRI8	Branch if one register is greater or equal to an encoded constant as unsigned
BGEZ	BRI12	Branch if a register is greater than or equal to zero
BLT	RRI8	Branch if one register is less than a register
BLTI	BRI8	Branch if one register is less than an encoded constant
BLTU	RRI8	Branch if one register is less than a register as unsigned
BLTUI	RRI8	Branch if one register is less than an encoded constant as unsigned
BLTZ	BRI12	Branch if a register is less than zero

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

Table 4–26. Core Architecture Instructions (continued)

Instruction[1]	Format	Definition
BNALL/BNONE	RRI8	Branch if some/all bits specified by a mask in a register are clear in another register
BNE	RRI8	Branch if a register does not equal a register
BNEI	RRI8	Branch if a register does not equal an encoded constant
BNEZ	BRI12	Branch if a register does not equal zero
CALL0	CALL	Call subroutine at PC plus offset, place return address in A0
CALLX0	CALLX	Call subroutine register specified location, place return address in A0
DSYNC/ESYNC	RRR	Wait for data memory/execution related changes to resolve
EXTUI	RRR	Extract field specified by immediates from a register
EXTW	RRR	Wait for any possible external ordering requirement (added in RA-2004.1)
ISYNC	RRR	Wait for instruction fetch related changes to resolve
J	CALL	Jump to PC plus offset
JX	CALLX	Jump to register specified location
L8UI	RRI8	Load zero extended byte
L16SI/L16UI	RRI8	Load sign/zero extended 16-bit quantity
L32I	RRI8	Load 32-bit quantity
L32R	RI16	Load literal at offset from PC (or from LITBASE with the Extended L32R Option)
MEMW	RRR	Wait for any possible memory ordering requirement
MOVEQZ	RRR	Move register if the contents of a register is zero
MOVGEZ	RRR	Move register if the contents of a register is greater than or equal to zero
MOVI	RRI8	Move a 12-bit immediate to a register
MOVLTZ	RRR	Move register if the contents of a register is less than zero
MOVNEZ	RRR	Move register if the contents of a register is not zero
NEG	RRR	Negate a register
NOP	RRR	No operation (added as a full instruction in RA-2004.1)
OR	RRR	Bitwise OR two registers
RET	CALLX	Subroutine return through A0
RSR.*	RSR	Read a Special Register
RSYNC	RRR	Wait for dispatch related changes to resolve
S8I/S16I/S32I	RRI8	Store byte/16-bit quantity/32-bit quantity
SLL/SLLI	RRR	Shift left logical by SAR/immediate
SRA/SRAI	RRR	Shift right arithmetic by SAR/immediate
SRC	RRR	Shift right combined by SAR with two registers as input and one as output

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

Table 4–26. Core Architecture Instructions (continued)

Instruction[1]	Format	Definition
SRL/SRLI	RRR	Shift right logical by SAR/immediate
SSA8B/SSA8L	RRR	Use low 2-bits of address register to prepare SAR for SRC assuming big/little endian
SSAI	RRR	Set SAR to immediate value
SSL/SSR	RRR	Set SAR from register for left/right shift
SUB	RRR	Subtract two registers
SUBX2/4/8	RRR	Subtract two registers with the un-negated one shifted left by one/two/three
WSR.*	RSR	Write a special register
XOR	RRR	Bitwise XOR two registers
XSR.*	RRR	Read and write a special register in an exchange (added in T1040)

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.3 Options for Additional Instructions

The options in this section have the primary function of adding new instructions to the processor's instruction set. The new instructions cover a variety of purposes including new architectural capabilities, higher performance on existing capabilities, and smaller code.

4.3.1 Code Density Option

This option adds 16-bit encodings of the most frequently used 24-bit instructions. When a 24-bit instruction can be encoded into a 16-bit form, the code-size savings is significant.

- Prerequisites: None
- Incompatible options: None
- Compatibility note: The additions made by this option were once considered part of the core architecture, thus compatibility with binaries for previous hardware might require the use of this option. Many available third-party software packages including some currently supported operating systems require the Code Density Option.

4.3.1.1 Code Density Option Architectural Additions

Table 4–27 shows this option's architectural additions.

Table 4–27. Code Density Option Instruction Additions

Instruction[1]	Format	Definition
ADD.N	RRRN	Add two registers (same as ADD instruction but with a 16-bit encoding).
ADDI.N	RRRN	Add register and immediate (-1 and 1..15).
BEQZ.N	RI16	Branch if register is zero with a 6-bit unsigned offset (forward only).
BNEZ.N	RI16	Branch if register is non-zero with a 6-bit unsigned offset (forward only).
BREAK.N[2]	RRRN	This instruction is the same as BREAK but with a 16-bit encoding.
L32I.N	RRRN	Load 32 bits, 4-bit offset
MOV.N	RRRN	Narrow move
MOVI.N	RI7	Load register with immediate (-32..95).
NOP.N	RRRN	This instruction performs no operation. It is typically used for instruction alignment.
RET.N	RRRN	The same as RET but with a 16-bit encoding.
RETW.N[3]	RRRN	The same as RETW but with a 16-bit encoding.
S32I.N	RRRN	Store 32 bits, 4-bit offset

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

[2] Exists only if the Debug Option described in Section 4.7.6 on page 197 is configured.

[3] Exists only if the Windowed Register Option described in Section 4.7.1 on page 180 is configured.

4.3.1.2 Branches

For some implementations, branches to an instruction that crosses a 32-bit memory boundary may suffer a small performance penalty. The compiler (or assembler) is expected to align performance-critical branch targets such that their byte address is 0 mod 4, 1 mod 4, or for 16-bit instructions, 2 mod 4. This can be accomplished either by converting some previous 16-bit-encoded instructions back to their 24-bit form, or by inserting a 16-bit NOP.N.

4.3.2 Loop Option

The Loop Option adds the ability for the processor to execute a zero-overhead loop where the number of iterations (not counting an early exit) can be determined prior to entering the loop. This capability is useful in digital signal processing applications where the overhead of a branch in a heavily used loop is unacceptable. A single loop instruction defines both the beginning and end of a loop, as well as a count of how many times the loop will execute.

- Prerequisites: None
- Incompatible options: None

- Compatibility note: The additions made by this option were once considered part of the core architecture, thus compatibility with binaries for previous hardware might require the use of this option. Many available third-party software packages including some currently supported operating systems require the Loop Option.

4.3.2.1 Loop Option Architectural Additions

Table 4–28 and Table 4–29 show this option's architectural additions.

Table 4–28. Loop Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
LBEG	1	32	Loop begin	R/W	0
LEND	1	32	Loop end	R/W	1
LCOUNT	1	32	Loop count	R/W	2

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 3–23 on page 46.

LBEG and LEND are undefined after processor reset. LCOUNT is initialized to zero after processor reset.

Table 4–29. Loop Option Instruction Additions

Instruction[1]	Format	Definition
LOOP	BRI8	Set up a zero-overhead loop by setting LBEG, LEND, and LCOUNT special registers.
LOOPGTZ	BRI8	Set up a zero-overhead loop by setting LBEG, LEND, and LCOUNT special registers. Skip loop if LCOUNT is not positive.
LOOPNEZ	BRI8	Set up a zero-overhead loop by setting LBEG, LEND, and LCOUNT special registers. Skip loop if LCOUNT is zero.

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.3.2.2 Restrictions on Loops

There is a restriction on instruction alignment for zero-overhead loops. The first instruction after the LOOP instruction, which begins at the address written to LBEG by the LOOP instruction, must be entirely contained within a naturally aligned, power of two sized unit of a particular size. That size is the next larger power of two equal to or greater than the instruction length, but not less than 4 bytes. Thus a 16-bit instruction, if it is the first in a loop, may be at 0 mod 4, 1 mod 4, or 2 mod 4. A 24-bit instruction, if it is the first in a loop, may be at 0 mod 4 or at 1 mod 4. As an example of a potential larger instruction, a 64-bit instruction must be aligned at 0 mod 8.

The last instruction of the loop must not be a call, ISYNC, WAITI, or RSR.LCOUNT. If the last instruction of the loop is a taken branch, then the value of LCOUNT is undefined. Thus, a taken branch may be used to exit the loop (in which case the value of LCOUNT is irrelevant), but not to iterate within the loop.

4.3.2.3 Loops Disabled During Exceptions

Loops are disabled when PS.EXCM is set in Xtensa Exception Architecture 2 and above. This prevents program code from maliciously or accidentally setting LEND to an address in an exception handler and then causing the exception, thereby transitioning to Ring 0 while retaining control of the processor.

4.3.2.4 Loopback Semantics

The processor includes the following to compute the PC of the next instruction:

```
if LCOUNT ≠ 0 and CLOOPENABLE and nextPC = LEND then
     LCOUNT ← LCOUNT − 1
     nextPC ← LBEG
endif
```

The semantics above have some non-obvious consequences. A taken branch to the address in LEND does not cause a transfer to LBEG. Thus a taken branch to the LEND instruction can be used to exit the loop prematurely. This is why a call instruction as the last instruction of a loop will not do the obvious thing (the return will branch to the LEND address and exit the loop). To conditionally begin the next loop iteration, a branch to a NOP before LEND may be used.

4.3.3 Extended L32R Option

The Extended L32R Option adds functionality to the standard L32R instruction. The standard L32R instruction has an offset that can reach as far as 256kB below the current PC. In the case where an instruction RAM approaches or exceeds 256kB in size, accessing literal data becomes much more difficult. This option is intended to ease the access to literal data by providing an optional separate literal base register.

- Prerequisites: None
- Incompatible options: MMU Option (page 158)

4.3.3.1 Extended L32R Option Architectural Additions

Table 4–30 shows this option’s architectural additions.

Table 4–30. Extended L32R Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
`LITBASE`	1	21	Literal base[2]	R/W	5

[1] Registers with a Special Register assignment are read and/or written with the `RSR`, `WSR`, and `XSR` instructions. See Table 3–23 on page 46.

[2] See Figure 4–7 on page 57 for the format of this register.

4.3.3.2 The Literal Base Register

The literal base (`LITBASE`) register contains 20 upper bits, which define the location of the literal base and one enable bit (En). When the enable bit is clear, the `L32R` instruction loads a literal at a negative offset from the PC. When the enable bit is set, the `L32R` instruction loads a literal at a negative offset from the address formed by the 20 upper bits of literal base and 12 lower bits of 12’h000. See the `L32R` instruction description in Chapter 6. Figure 4–7 shows the `LITBASE` register format.

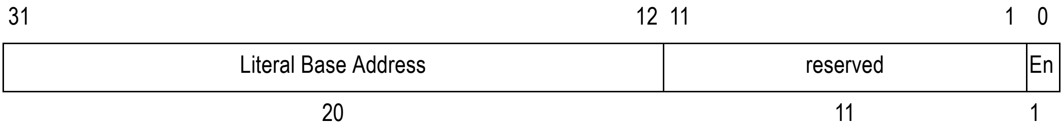


Figure 4–7. LITBASE Register Format

The enable bit of the literal base register is cleared after reset. The remaining bits are undefined after reset.

4.3.4 16-bit Integer Multiply Option

This option provides two instructions that perform 16×16 multiplication, producing a 32-bit result. It is typically useful for digital signal processing (DSP) algorithms that require 16 bits or less of input precision (32 bits of input precision is provided by the 32-bit Integer Multiply Option) and do not require more than 32-bit accumulation (as provided by the MAC16 Option). Because a 16×16 multiplier is one-fourth the area of a 32×32 multiplier, this option is less costly than the 32-bit Integer Multiply Option. Because it lacks an accumulator and data registers, it is less costly than the MAC16 Option.

- Prerequisites: None
- Incompatible options: None
- See Also "MAC16 Option" on page 60 and "32-bit Integer Multiply Option" on page 58

4.3.4.1 16-bit Integer Multiply Option Architectural Additions

Table 4–31 shows this option's architectural additions. There are no configuration parameters associated with the MUL16 Option and no additional processor state.

Table 4–31. 16-bit Integer Multiply Option Instruction Additions

Instruction[1]	Format	Definition
`MUL16S`	RRR	Signed 16×16 multiplication of the least-significant 16 bits of `AR[s]` and `AR[t]`, with the 32-bit product written to `AR[r]`
`MUL16U`	RRR	Unsigned 16×16 multiplication of the least-significant 16 bits of `AR[s]` and `AR[t]`, with the 32-bit product written to `AR[r]`

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.3.5 32-bit Integer Multiply Option

This option provides instructions that implement 32-bit integer multiplication as instructions. This provides single instruction targets for the multiplication operators of programming languages such as C. When this option is not enabled, the Xtensa compiler uses subroutine calls to implement 32-bit integer multiplication. Note that various algorithms may be used to implement multiplication, and some hardware implementations may be slower than the software implementations for some operand values. Implementations may allow a choice of algorithms through configuration parameters to optimize among area, speed, and other characteristics.

There is one sub-option within this option: Mul32High. It controls whether the `MULSH` and `MULUH` instructions are included or not. For some implementations, generating the high 32 bits of the product requires additional hardware, and so disabling this sub-option may reduce cost.

- Prerequisites: None
- Incompatible options: None
- See Also: "MAC16 Option" on page 60 and "16-bit Integer Multiply Option" on page 57

4.3.5.1 32-bit Integer Multiply Option Architectural Additions

Table 4–32 and Table 4–33 show this option's architectural additions. This option adds no new processor state.

Table 4–32. 32-bit Integer Multiply Option Processor-Configuration Additions

Parameter	Description	Valid Values
Mul32High	Determines whether the MULSH and MULUH instructions are included	0 or 1
MulAlgorithm	Determines the multiplication algorithm employed	Implementation-dependent

Table 4–33. 32-Bit Integer Multiply Instruction Additions

Instruction[1]	Format	Definition
MULL	RRR	Multiply low (return least-significant 32 bits of product)
MULUH[2]	RRR	Multiply unsigned high (return most-significant 32 bits of product)
MULSH[2]	RRR	Multiply signed high (return most-significant 32 bits of product)

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

[2] These instructions are part of the Mul32High sub-option of 32-bit Integer Multiply Option.

4.3.6 32-bit Integer Divide Option

This option provides instructions that implement 32-bit integer division and remainder operations. When this option is not enabled, the Xtensa compiler uses subroutine calls to implement division and remainder. Note that various algorithms may be used to implement these instructions, and some hardware implementations may be slower than the software implementations for some operand values.

- Prerequisites: None
- Incompatible Options: None

4.3.6.1 32-bit Integer Divide Option Architectural Additions

Table 4–34 through Table 4–36 show this option's architectural additions. This option adds no new processor state. This option does add a new exception, Integer Divide by Zero, which is raised when the divisor operand of a QUOS, QUOU, REMS, or REMU instruction contains zero.

Table 4–34. 32-bit Integer Divide Option Processor-Configuration Additions

Parameter	Description	Valid Values
DivAlgorithm	Determines the division algorithm employed	Implementation-dependent

Table 4–35. 32-bit Integer Divide Option Exception Additions

Exception	Description	EXCCAUSE value
IntegerDivideByZero	Exception raised when divisor is zero	6

Table 4–36. 32-bit Integer Divide Option Instruction Additions

Instruction[1]	Format	Definition
QUOS	RRR	Quotient Signed (divide giving 32-bit quotient)
QUOU	RRR	Quotient Unsigned (divide giving 32-bit quotient)
REMS	RRR	Remainder Signed (divide giving 32-bit remainder)
REMU	RRR	Remainder Unsigned (divide giving 32-bit remainder)

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243

4.3.7 MAC16 Option

The MAC16 Option adds multiply-accumulate functions that are useful in DSP and other media-processing operations. The option adds a 40-bit accumulator (ACC), four 32-bit data registers (MR[n]), and 72 instructions.

The multiplier operates on two 16-bits operands from either the address registers (AR) or MAC16 registers (MR). Each operand may be taken from either the low or high half of a register. The result of the operation is placed in the 40-bit accumulator. The MR registers and the low 32 bits and high 8 bits of the accumulator are readable and writable with the RSR, WSR, and XSR instructions. MR[0] and MR[1] can be used as the first multiplier input, and MR[2] and MR[3] can be used as the second multiplier input. Four of the 72 added instructions can load the MR registers with 32-bit values from memory in parallel with multiply-accumulate operations.

The accumulator (ACC) and data registers (MR) are undefined after reset.

- Prerequisites: None
- Incompatible options: None

4.3.7.1 MAC16 Option Architectural Additions

Table 4–37 and Table 4–38 show this option’s architectural additions.

Table 4–37. MAC16 Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
ACCLO	1	32	Accumulator low	R/W	16
ACCHI	1	8	Accumulator high	R/W	17
MR[0][2]	1	32	MAC16 register 0 (m0 in assembler)	R/W	32
MR[1][2]	1	32	MAC16 register 1 (m1 in assembler)	R/W	33
MR[2][2]	1	32	MAC16 register 2 (m2 in assembler)	R/W	34
MR[3][2]	1	32	MAC16 register 3 (m3 in assembler)	R/W	35

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 3–23 on page 46.

[2] These registers are known as MR[0..3] in hardware and as m0..3 in the software.

Table 4–38. MAC16 Option Instruction Additions

Instruction[1, 2]	Definition[3]
LDDEC	Load MAC16 data register (MR) with auto decrement
LDINC	Load MAC16 data register (MR) with auto increment
MUL.AA.*qq*	Signed multiply of two address registers
MUL.AD.*qq*	Signed multiply of an address register and a MAC16 data register
MUL.DA.*qq*	Signed multiply of a MAC16 data register and an address register
MUL.DD.*qq*	Signed multiply of two MAC16 data registers
MULA.AA.*qq*	Signed multiply-accumulate of two address registers
MULA.AD.*qq*	Signed multiply-accumulate of an address register and a MAC16 data register
MULA.DA.*qq*	Signed multiply-accumulate of a MAC16 data register and an address register
MULA.DD.*qq*	Signed multiply-accumulate of two MAC16 data registers
MULS.AA.*qq*	Signed multiply/subtract of two address registers
MULS.AD.*qq*	Signed multiply/subtract of an address register and a MAC16 data register
MULS.DA.*qq*	Signed multiply/subtract of a MAC16 data register and an address register
MULS.DD.*qq*	Signed multiply/subtract of two MAC16 data registers
MULA.DA.*qq*.LDDEC	Signed multiply-accumulate of a MAC16 data register and an address register, and load a MAC16 data register with auto decrement
MULA.DA.*qq*.LDINC	Signed multiply-accumulate of a MAC16 data register and an address register, and load a MAC16 data register with auto increment

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

[2] The *qq* opcode parameter indicates (by HH, HL, LH, or LL) whether the operands are taken from the Low or High 16-bit half of the AR or MR registers. The first *q* represents the location of the first operand; the second *q* represents the location of the second operand.

[3] The destination for all product and accumulate results is the MAC16 accumulator

Table 4–38. MAC16 Option Instruction Additions (continued)

Instruction[1, 2]	Definition[3]
MULA.DD.*qq*.LDDEC	Signed multiply-accumulate of two MAC16 data registers, and load a MAC16 data register with auto decrement
MULA.DD.*qq*.LDINC	Signed multiply-accumulate of two MAC16 data registers, and load a MAC16 data register with auto increment
UMUL.AA.*qq*	Unsigned multiply of two address registers

1. These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.
2. The *qq* opcode parameter indicates (by HH, HL, LH, or LL) whether the operands are taken from the Low or High 16-bit half of the AR or MR registers. The first *q* represents the location of the first operand; the second *q* represents the location of the second operand.
3. The destination for all product and accumulate results is the MAC16 accumulator

4.3.7.2 Use With CLAMPS Instruction

The CLAMPS instruction, implemented with the Miscellaneous Operations Option, is useful in conjunction with the MAC16 Option. It allows clamping results to 16 bits before storing to memory.

4.3.8 Miscellaneous Operations Option

These instructions can be individually enabled in groups to provide computational capability required by a few applications.

- Prerequisites: None
- Incompatible options: None

4.3.8.1 Miscellaneous Operations Option Architectural Additions

Table 4–39 and Table 4–40 show this option’s architectural additions.

Table 4–39. Miscellaneous Operations Option Processor-Configuration Additions

Parameter	Description	Valid Values
InstructionCLAMPS	Enable the signed clamp instruction: CLAMPS	0 or 1
InstructionMINMAX	Enable the minimum and maximum value instructions: MIN, MAX, MINU, MAXU	0 or 1
InstructionNSA	Enabled the normalization shift amount instructions: NSA, NSAU	0 or 1
InstructionSEXT	Enable the sign extend instruction: SEXT	0 or 1

Table 4–40. Miscellaneous Operations Instruction Additions

Instruction[1]	Format	Definition

CLAMPS	RRR	Clamp to signed power of two range
		sign ← AR[s]$_{31}$
		AR[r] ← if AR[s]$_{30..(t+7)}$ = sign$^{24–t}$
		        then AR[s]
		        else sign$^{(25–t)}$ $\Vert$ (not sign)$^{t+7}$

MAX	RRR	Maximum value signed AR[r] ← if AR[s] < AR[t] then AR[t] else AR[s]

MAXU	RRR	Maximum value unsigned
		AR[r] ← if (0$\vert\vert$AR[s]) < (0$\vert\vert$AR[t])
		        then AR[t]
		        else AR[s]

MIN	RRR	Minimum value signed AR[r] ← if AR[s] < AR[t] then AR[s] else AR[t]

MINU	RRR	Minimum value unsigned
		AR[r] ← if (0$\vert\vert$AR[s]) < (0$\vert\vert$AR[t])
		        then AR[s]
		        else AR[t]

NSA	RRR	Normalization shift amount signed AR[r] ← nsa[1](AR[s]$_{31}$, AR[s]) NSA returns the number of contiguous bits in the most significant end of AR[s] that are equal to the sign bit (not counting the sign bit itself), or 31 if AR[s] = 0 or AR[s] = -1. The result may be used as a left shift amount such that the result of SLL on AR[s] will have bit31 ≠ bit30 (if AR[s] ≠ 0).
NSAU	RRR	Normalization shift amount unsigned AR[r] ← nsa[1](0, AR[s]) NSAU returns the number of contiguous zero bits in the most significant end of AR[s], or 32 if AR[s] = 0. The result may be used as a left shift amount such that the result of SLL on AR[s] will have bit31 ≠ 0 (if AR[s] ≠ 0).

SEXT	RRR	Sign extend
		sign ← AR[s]$_{t+7}$
		AR[r] ← sign$^{(24–t)}$ $\vert\vert$ AR[s]$_{t+7..0}$

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.3.9 Coprocessor Option

A coprocessor is a combination of additional state, instructions and logic that operates on that state, including moves and the setting of Booleans for branch true/false operations. The Coprocessor Option is general in nature: it adds state that is shared by all coprocessors. After the Coprocessor Option is added, specific coprocessors, such as the Floating-Point Coprocessor Option, can be added, along with system-specific instructions for coprocessor operations.

- Prerequisites: Exception Option (page 82)
- Incompatible options: None

4.3.9.1 Coprocessor Option Architectural Additions

Table 4–41 and Table 4–42 show this option's architectural additions.

Table 4–41. Coprocessor Option Exception Additions

Exception	Description	EXCCAUSE value
Coprocessor0Disabled	Coprocessor 0 instruction while cp0 disabled	32
Coprocessor1Disabled	Coprocessor 1 instruction while cp1 disabled	33
Coprocessor2Disabled	Coprocessor 2 instruction while cp2 disabled	34
Coprocessor3Disabled	Coprocessor 3 instruction while cp3 disabled	35
Coprocessor4Disabled	Coprocessor 4 instruction while cp4 disabled	36
Coprocessor5Disabled	Coprocessor 5 instruction while cp5 disabled	37
Coprocessor6Disabled	Coprocessor 6 instruction while cp6 disabled	38
Coprocessor7Disabled	Coprocessor 7 instruction while cp7 disabled	39

Table 4–42. Coprocessor Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
CPENABLE	1	8	Coprocessor enable bits	R/W	224

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 3–23 on page 46.

4.3.9.2 Coprocessor Context Switch

RUR and WUR are not created by the Coprocessor Option, but rather by TIE language constructs. They provide a uniform way for reading and writing miscellaneous state added via the TIE language. The TIE user_register construct associates TIE state registers with RUR/WUR register numbers in 32-bit quantities. RUR reads 32 bits of TIE state into an address register, and WUR writes 32 bits to a TIE state register from an address register. The ISA does not define the result of additional bits read by RUR when fewer than 32 bits of TIE state are associated with the user register.

The TIE compiler automatically generates for each coprocessor the assembly code to save the state associated with a coprocessor to memory and to restore coprocessor state from memory.

Tensilica reserves user register numbers for RUR and WUR in the range 192 to 255.

The CPENABLE register allows a “lazy” context switch of the coprocessor state. Any instruction that references coprocessor *n* state (not including the shared Boolean registers) when that coprocessor’s enable bit (bit *n*) is clear raises a Coprocessor*n*Disabled exception. CPENABLE can be cleared on context switch, and the exception used to unload the previous task’s coprocessor state and load the current task’s. The appropriate CPENABLE bit is then set by the exception handler, which then returns to execute the coprocessor instruction. An RSYNC instruction must be executed after writing CPENABLE before executing any instruction that references state controlled by the changed bits of CPENABLE. This register is undefined after reset.

If a single instruction references state from more than one coprocessor not enabled in CPENABLE, then one of Coprocessor*n*Disabled exceptions is raised. The prioritization among multiple Coprocessor*n*Disabled exceptions is implementation-specific.

4.3.10 Boolean Option

This option makes a set of Boolean registers available, along with branches and other operations that refer to them. Multiple coprocessors and other TIE language extensions can use this set.

- Prerequisites: None
- Incompatible options: None

4.3.10.1 Boolean Option Architectural Additions

Table 4–43 and Table 4–44 show this option’s architectural additions.

Table 4–43. Boolean Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
BR[2]	16	1	Boolean registers	R/W	4

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 3–23 on page 46.

[2] This register is known as Special Register BR or as individual Boolean bits b0..15.

Table 4–44. Boolean Option Instruction Additions

Instruction[1]	Format	Definition
ALL4	RRR	4-Boolean and reduction (result is 1 if all of the 4 Booleans are true)
ALL8	RRR	8-Boolean and reduction (result is 1 if all of the 8 Booleans are true)
ANDB	RRR	Boolean and
ANDBC	RRR	Boolean and with complement
ANY4	RRR	4-Boolean or reduction (result is 1 if any of the 4 Booleans is true)
ANY8	RRR	8-Boolean or reduction (result is 1 if any of the 8 Booleans is true)
BF	RRI8	Branch if Boolean false
BT	RRI8	Branch if Boolean true
MOVF	RRR	Conditional move if false
MOVT	RRR	Conditional move if true
ORB	RRR	Boolean or
ORBC	RRR	Boolean or with complement
XORB	RRR	Boolean exclusive or

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.3.10.2 Booleans

A coprocessor test or comparison produces a Boolean result. The Boolean Option provides 16 single-bit Boolean registers for storing the results of coprocessor comparisons for testing in conditional move and branch instructions. Boolean logic may replace branches in some situations. Compared to condition codes used by other ISAs, these Booleans eliminate the bottleneck of having only a single place to store comparison results. It is possible, for example, to do multiple comparisons before the comparison results are used. For Single-Instruction Multiple-Data (SIMD) operations, Booleans provide up to 16 simultaneous compare results and conditionals.

Boolean-producing instructions generate only one sense of the condition (for example, = but not ≠); all Boolean uses allow for complementing of the Boolean. Multiple Booleans may be combined into a single Boolean using the ANY4, ALL4, and so forth instructions. For example, this is useful after a SIMD comparison to test if any or all of the elements satisfy the test, such as testing if any byte of a word is zero. ANY2 and ALL2 instructions are not provided; ANDB and ORB provide this functionality given bs+0 and bs+1 as arguments.

The Boolean registers are undefined after reset.

The Boolean registers are accessible from C using the `xtbool`, `xtbool2`, `xtbool4`, `xtbool8`, and `xtbool16` data types. See the *Xtensa C and C++ Compiler User's Guide* for details.

4.3.11 Floating-Point Coprocessor Option

The Floating-Point Coprocessor Option adds the logic and architectural components needed for IEEE754 single-precision floating-point operations. These operations are useful for DSP that requires >16 bits of precision, such as audio compression and decompression. Also, DSP algorithms for less precise data are more easily coded using floating-point, and good performance is obtainable when programming in languages such as C.

- Prerequisites: Coprocessor Option (page 63) and Boolean Option (page 65)
- Incompatible options: None

4.3.11.1 Floating-Point Coprocessor Option Architectural Additions

Table 4–45 through Table 4–46 show this option's architectural additions.

Table 4–45. Floating-Point Coprocessor Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Register Number[1]
`FR`	16	32	Floating-point register	R/W	-
`FCR`	1	32	Floating-point control register	R/W	User 232
`FSR`	1	32	Floating-point status register	R/W	User 233

[1] See Table 3–23 on page 46.

Table 4–46. Floating-Point Coprocessor Option Instruction Additions

Instruction[1]	Format	Definition
`ABS.S`	RRR	Single-precision absolute value
`ADD.S`	RRR	Single-precision add
`CEIL.S`	RRR	Single-precision floating-point to signed integer conversion with round to +∞
`FLOAT.S`	RRR	Signed integer to single-precision floating-point conversion (current rounding mode)
`FLOOR.S`	RRR	Single-precision floating-point to signed integer conversion with round to -∞
`LSI`	RRI8	Load single-precision immediate
`LSIU`	RRI8	Load single-precision immediate with base update

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

Table 4–46. Floating-Point Coprocessor Option Instruction Additions (continued)

Instruction[1]	Format	Definition
LSX	RRR	Load single-precision indexed
LSXU	RRR	Load single-precision indexed with base update
MADD.S	RRR	Single-precision multiply-add
MOV.S	RRR	Single-precision move
MOVEQZ.S	RRR	Single-precision move if equal to zero
MOVF.S	RRR	Single-precision move if Boolean condition false
MOVGEZ.S	RRR	Single-precision move if greater than or equal to zero
MOVLTZ.S	RRR	Single-precision move if less than zero
MOVNEZ.S	RRR	Single-precision move if not equal to zero
MOVT.S	RRR	Single-precision move if Boolean condition true
MSUB.S	RRR	Single-precision multiply-subtract
MUL.S	RRR	Single-precision multiply
NEG.S	RRR	Single-precision negate
OEQ.S	RRR	Single-precision compare equal
OLE.S	RRR	Single-precision compare less than or equal
OLT.S	RRR	Single-precision compare less than
RFR	RRR	Read floating-point register (FR to AR)
ROUND.S	RRR	Single-precision floating-point to signed integer conversion with round to nearest
SSI	RRI8	Store single-precision immediate
SSIU	RRI8	Store single-precision immediate with base update
SSX	RRR	Store single-precision indexed
SSXU	RRR	Store single-precision indexed with base update
SUB.S	RRR	Single-precision subtract
TRUNC.S	RRR	Single-precision floating-point to signed integer conversion with round to 0
UEQ.S	RRR	Single-precision compare unordered or equal
UFLOAT.S	RRR	Unsigned integer to single-precision floating-point conversion (current rounding mode)
ULE.S	RRR	Single-precision compare unordered or less than or equal
ULT.S	RRR	Single-precision compare unordered or less than
UN.S	RRR	Single-precision compare unordered
UTRUNC.S	RRR	Single-precision floating-point to unsigned integer conversion with round to 0
WFR	RRR	Write floating-point register (AR to FR)

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.3.11.2 Floating-Point Representation

The primary floating-point data type is IEEE754 single-precision:

31	30 23	22 0
s	exp	fraction
1	8	23

The other data format is a signed, 32-bit integer used by the `FLOAT.S`, `TRUNC.S`, `ROUND.S`, `FLOOR.S`, and `CEIL.S` instructions.

IEEE754 uses a sign-magnitude format, with a 1-bit sign, an 8-bit exponent with bias 127, and a 24-bit significand formed from 23 stored bits representing the binary digits to the right the binary point, and an implicit bit to the left of the binary point (0 if exponent is zero, 1 if exponent is non-zero). Thus, the value of the number is:

$$(-1)^{\mathtt{s}} \times 2^{\mathtt{exp}-127} \times \mathtt{implicit.fraction}$$

Thus, the representation for 1.0 is 0x3F800000, with a sign of 0, exp of 127, a zero fraction, and an implicit 1 to the left of the binary point.

The Xtensa ISA includes IEEE754 signed-zero, infinity, quiet NaN, and sub-normal representations and processing rules. The ISA does not include IEEE754 signaling NaNs or exceptions. Integer ⇔ floating-point conversions include a binary scale factor to make conversion into and out of fixed-point formats faster.

4.3.11.3 Floating-Point State

Table 4–45 summarizes the processor state added by the floating-point coprocessor. The FR register file consists of 16 registers of 32 bits each and is used for all data computation. Load and store instructions transfer data between the FR's and memory. The FCR register file has one field that may be changed at run-time to control the operation of various instructions. Table 4–47 lists FCR fields and their associated meanings. The format of FCR is

31 12	11 7	6	5	4	3	2	1 0
reserved	ignore	V	Z	O	U	I	RM
20	5	1	1	1	1	1	2

Table 4–47. FCR fields

FCR Field	Meaning
RM	Rounding mode 0 → round to nearest 1 → round toward 0 (TRUNC) 2 → round toward +∞ (CEIL) 3 → round toward –∞ (FLOOR)
I	Inexact exception enable (0 → disabled, 1 → enabled)
U	Underflow exception enable (0 → disabled, 1 → enabled)
O	Overflow exception enable (0 → disabled, 1 → enabled)
Z	Divide-by-zero exception enable (0 → disabled, 1 → enabled)
V	Invalid exception enable (0 → disabled, 1 → enabled)
ignore	Reads as 0, ignored on write
reserved	Reads back last value written. Non-zero values cause a floating-point exception on any floating-point instruction (see Section 4.3.11.4)

The `FSR` register file provides the status flags required by IEEE754. These flags are set by any operation that raises a non-enabled exception (see Section 4.3.11.4). Enabled exceptions abort the operation with a floating-point exception and the flags are not written:

31 12	11	10	9	8	7	6 0
reserved	V	Z	O	U	I	ignore
20	1	1	1	1	1	7

Table 4–48. FSR fields

FSR Field	Meaning
I	Inexact exception flag
U	Underflow exception flag
O	Overflow exception flag
Z	Divide-by-zero flag
V	Invalid exception flag
ignore	Reads as 0, ignored on write
reserved	Reads back last value written. Non-zero values cause a floating-point exception on any floating-point instruction (see Section 4.3.11.4)

Most architectures have a combined floating-point control and status register, instead of separate registers. In high-performance pipelines, this combination can compromise performance, as reads and writes must access all bits, even ones that are not required by the program. Xtensa's FCR may be read and written without waiting for the results of pending floating-point operations. Writes to FCR affect subsequent floating-point operations, but there is usually little performance cost from this dependency. Only reads of FSR need cause a significant pipeline interlock.

FCR and FSR are organized to allow implementation with a single 32-bit physical register. The separate register numbers affect only the bits read and written of this underlying physical register. It is also possible for software to bitwise logical OR the RUR's of FCR and FSR to create the appearance of a single register and to write this combined value to FCR and FSR.

The reserved bits of FCR and FSR must store the last value written, but if that value is non-zero, this causes all floating-point operations to raise a floating-point exception. This allows future extensions to define additional control values that if used in earlier implementations, can be emulated in software.

4.3.11.4 Floating-Point Exceptions

Current implementations neither raise exceptions enabled by FCR bits nor set flag bits in FSR. They also do not raise an exception when one of the reserved bits of FCR or FSR is non-zero.

4.3.11.5 Floating-Point Instructions

The floating-point instructions are defined in Table 4–49 and Table 4–50. The instructions operate on data in the floating-point register file, which consists of 16 32-bit registers.

The floating-point ISA requires a triple read-port FR register file for the MADD.S and MSUB.S operations.

Table 4–49. Floating-Point Coprocessor Option Load/Store Instructions

Instruction[1]	Format	Definition
LSI	RRI8	Load single-precision immediate $\text{vAddr} \leftarrow \text{AR[s]} + (0^{22} \Vert \text{imm8} \Vert 0^{2})$ FR[t] ← Load32(vAddr)

LSIU	RRI8	Load single-Precision Immediate with Base Update
		$\text{vAddr} \leftarrow \text{AR[s]} + (0^{22} \Vert \text{imm8} \Vert 0^{2})$
		FR[t] ← Load32(vAddr)
		AR[s] ← vAddr

LSX	RRR	Load single-Precision Indexed
		vAddr ← AR[s] + AR[t]
		FR[t] ← Load32(vAddr)

LSXU	RRR	Load single-Precision Indexed with Base Update
		vAddr ← AR[s] + AR[t]
		FR[t] ← Load32(vAddr)
		AR[s] ← vAddr

SSI	RRI8	Store single-Precision Immediate $\text{vAddr} \leftarrow \text{AR[s]} + (0^{22} \Vert \text{imm8} \Vert 0^{2})$ Store32 (vAddr, FR[t])
SSIU	RRI8	Store single-Precision Immediate with Base Update $\text{vAddr} \leftarrow \text{AR[s]} + (0^{22} \Vert \text{imm8} \Vert 0^{2})$ Store32 (vAddr, FR[t]) AR[s] ← vAddr
SSX	RRR	Store single-Precision Indexed vAddr ← AR[s] + AR[t] Store32 (vAddr, FR[r])

SSXU	RRR	Store single-Precision Indexed with Base Update
		vAddr ← AR[s] + AR[t]
		Store32 (vAddr, FR[r])
		AR[s] ← vAddr

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243

Table 4–50. Floating-Point Coprocessor Option Operation Instructions

Instruction[1]	Format	Definition
ABS.S	RRR	Single-precision absolute value FR[r] ← abs$_s$(FR[s])
ADD.S	RRR	Single-precision add FR[r] ← FR[s] $+_s$ FR[t]
CEIL.S	RRR	Scale and convert single-precision to integer, round to +∞ AR[r] ← ceil$_s$(FR[s] $\times_s$ pow$_s$(2.0,t))

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

Table 4–50. Floating-Point Coprocessor Option Operation Instructions (continued)

Instruction[1]	Format	Definition
FLOAT.S	RRR	Convert signed integer to single-precision and scale $\mathtt{FR[r]} \leftarrow \mathtt{float}_{\mathtt{s}}\mathtt{(AR[s])} \times_{\mathtt{s}} \mathtt{pow}_{\mathtt{s}}\mathtt{(2.0,-t)}$
FLOOR.S	RRR	Scale and convert single-precision to integer, round to –∞ $\mathtt{AR[r]} \leftarrow \mathtt{floor}_{\mathtt{s}}\mathtt{(FR[s]} \times_{\mathtt{s}} \mathtt{pow}_{\mathtt{s}}\mathtt{(2.0,t))}$
MADD.S	RRR	Single-precision multiply/add $\mathtt{FR[r]} \leftarrow \mathtt{FR[r]} +_{\mathtt{s}} \mathtt{(FR[s]} \times_{\mathtt{s}} \mathtt{FR[t])}$
MOV.S	RRR	Single-precision move $\mathtt{FR[r]} \leftarrow \mathtt{FR[s]}$
MOVEQZ.S	RRR	Single-precision conditional move if equal to zero $\mathtt{if\ AR[t]} = \mathtt{0}^{32}\ \mathtt{then\ FR[r]} \leftarrow \mathtt{FR[s]\ endif}$
MOVF.S	RRR	Single-precision conditional move if false $\mathtt{if\ BR}_{\mathtt{t}} = \mathtt{0\ then\ FR[r]} \leftarrow \mathtt{FR[s]\ endif}$
MOVGEZ.S	RRR	Single-precision conditional move if greater than or equal to zero $\mathtt{if\ AR[t]}_{\mathtt{31}} = \mathtt{0\ then\ FR[r]} \leftarrow \mathtt{FR[s]\ endif}$
MOVLTZ.S	RRR	Single-precision conditional move if less than zero $\mathtt{if\ AR[t]}_{\mathtt{31}} \neq \mathtt{0\ then\ FR[r]} \leftarrow \mathtt{FR[s]\ endif}$
MOVNEZ.S	RRR	Single-precision conditional move if not equal to zero $\mathtt{if\ AR[t]} \neq \mathtt{0}^{32}\ \mathtt{then\ FR[r]} \leftarrow \mathtt{FR[s]\ endif}$
MOVT.S	RRR	Single-precision conditional move if true $\mathtt{if\ BR}_{\mathtt{t}} \neq \mathtt{0\ then\ FR[r]} \leftarrow \mathtt{FR[s]\ endif}$
MSUB.S	RRR	Single-precision multiply/subtract $\mathtt{FR[r]} \leftarrow \mathtt{FR[r]} -_{\mathtt{s}} \mathtt{(FR[s]} \times_{\mathtt{s}} \mathtt{FR[t])}$
MUL.S	RRR	Single-precision multiply $\mathtt{FR[r]} \leftarrow \mathtt{FR[s]} \times_{\mathtt{s}} \mathtt{FR[t]}$
NEG.S	RRR	Single-precision negate $\mathtt{FR[r]} \leftarrow -_{\mathtt{s}}\ \mathtt{FR[s]}$
OEQ.S	RRR	Single-precision compare equal $\mathtt{BR}_{\mathtt{r}} \leftarrow \mathtt{FR[s]\ OEQ}_{\mathtt{s}}\ \mathtt{FR[t];}$
OLE.S	RRR	Single-precision compare less than or equal $\mathtt{BR}_{\mathtt{r}} \leftarrow \mathtt{FR[s]\ OLE}_{\mathtt{s}}\ \mathtt{FR[t];}$
OLT.S	RRR	Single-precision compare less than $\mathtt{BR}_{\mathtt{r}} \leftarrow \mathtt{FR[s]\ OLT}_{\mathtt{s}}\ \mathtt{FR[t];}$
RFR	RRR	Move from FR to AR $\mathtt{AR[r]} \leftarrow \mathtt{FR[s]}$
ROUND.S	RRR	Scale and convert single-precision to integer, round to nearest $\mathtt{AR[r]} \leftarrow \mathtt{round}_{\mathtt{s}}\mathtt{(FR[s]} \times_{\mathtt{s}} \mathtt{pow}_{\mathtt{s}}\mathtt{(2.0,t))}$
SUB.S	RRR	Single-precision subtract $\mathtt{FR[r]} \leftarrow \mathtt{FR[s]} -_{\mathtt{s}} \mathtt{FR[t]}$

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

Table 4–50. Floating-Point Coprocessor Option Operation Instructions (continued)

Instruction[1]	Format	Definition
TRUNC.S	RRR	Scale and convert single-precision to signed integer, round to 0 $\texttt{AR[r]} \leftarrow \texttt{trunc}_S\texttt{(FR[s]} \times_S \texttt{pow}_S\texttt{(2.0,t))}$
UEQ.S	RRR	Single-precision compare unordered or equal $\texttt{BR}_r \leftarrow \texttt{FR[s] UEQ}_S\texttt{ FR[t];}$
UFLOAT.S	RRR	Convert unsigned integer to single-precision and scale $\texttt{FR[r]} \leftarrow \texttt{ufloats(AR[s])} \times_S \texttt{pow}_S\texttt{(2.0,-t))}$
ULE.S	RRR	Single-precision compare unordered or less than or equal $\texttt{BR}_r \leftarrow \texttt{FR[s] ULE}_S\texttt{ FR[t];}$
ULT.S	RRR	Single-precision compare unordered or less than $\texttt{BR}_r \leftarrow \texttt{FR[s] ULT}_S\texttt{ FR[t];}$
UN.S	RRR	Single-precision compare unordered $BR_r \leftarrow$ FR[s] UNs FR[t];
UTRUNC.S	RRR	Scale and convert single-precision to unsigned integer, round to 0 $\texttt{AR[r]} \leftarrow \texttt{utrunc}_S\texttt{(FR[s]} \times_S \texttt{pow}_S\texttt{(2.0,t))}$
WFR	RRR	Move from AR to FR FR[r] ← AR[s]

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.3.12 Multiprocessor Synchronization Option

When multiple processors are used in a system, some sort of communication and synchronization between processors is required. (Note that multiprocessor synchronization is distinct from pipeline synchronization between instructions as represented by the ISYNC, RSYNC, ESYNC, and DSYNC instructions, despite the name similarity). In some cases, self-synchronizing communication, such as input and output queues, is used. In other cases, a shared memory model is used for communication, and it is necessary to provide instruction-set support for synchronization because shared memory does not provide the required semantics. The Multiprocessor Synchronization Option is designed for this shared memory case.

- Prerequisites: None
- Incompatible Options: None

4.3.12.1 Memory Access Ordering

The Xtensa ISA requires that valid programs follow a simplified version of the Release Consistency model of memory access ordering. Xtensa implementations may perform ordinary load and store operations to non-overlapping addresses in any order. Loads and stores to overlapping addresses on a single processor must be executed in program order. This flexibility is appropriate because most memory accesses require only these semantics and some implementations may be able to execute programs significantly faster by exploiting non-program order memory access. While these semantics are appropriate for most loads and stores, order does matter when synchronizing between processors. Xtensa's Multiprocessor Synchronization Option therefore augments ordinary loads and stores with *acquire* and *release* operations, which are respectively loads and stores with more constrained memory ordering semantics relative to each other and relative to ordinary loads and stores.

The Xtensa version of Release Consistency is adapted from *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors* by Gharachorloo et. al. in the Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990, from which the following three definitions are directly borrowed:

- A load by processor *i* is considered *performed with respect to processor k* at a point in time when the issuing of a store to the same address by processor k cannot affect the value returned by the load.
- A store by processor i is considered *performed with respect to processor k* at a point in time when an issued load to the same address by processor k returns the value defined by this store (or a subsequent store to the same location).
- An access is *performed* when it is performed with respect to all processors.

Using these definitions, Xtensa places the following requirements on memory access:

- Before an ordinary load or store access is allowed to perform with respect to any other processor, all previous *acquire* accesses must be performed, and
- Before a *release* access is allowed to perform with respect to any other processor, all previous ordinary load, store, acquire, and release accesses must be performed, and
- Before an acquire is allowed to perform with respect to any other processor, all previous acquire accesses must be performed.

Many Xtensa implementations will adopt stricter memory orderings for simplicity. However, programs should not rely on any stricter memory ordering semantics than those specified here.

4.3.12.2 Multiprocessor Synchronization Option Architectural Additions

Table 4–51 shows this option's architectural additions.

Table 4–51. Multiprocessor Synchronization Option Instruction Additions

Instruction[1]	Format	Definition
L32AI	RRI8	Load 32-bit acquire (8-bit shifted offset) This load will perform before any subsequent loads, stores, or acquires are performed. It is typically used to test the synchronization variable protecting a critical region (for example, to acquire a lock).
S32RI	RRI8	Store 32-bit release (8-bit shifted offset) All prior loads, stores, acquires, and releases will be performed before this store is performed. It is typically used to write a synchronization variable to indicate that this processor is no longer in a critical region (for example, to release a lock).

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.3.12.3 Inter-Processor Communication with the L32AI and S32RI Instructions

L32AI and S32RI are 32-bit load and store instructions with acquire and release semantics. These instructions are useful for controlling the ordering of memory references in multiprocessor systems, where different memory locations may be used for synchronization and data, so that precise ordering between synchronization references must be maintained. Other load and store instructions may be executed by processor implementations in any order that produces the same uniprocessor result.

The MEMW instruction is somewhat similar in that it enforces load and store ordering, but is less selective. MEMW is intended for implementing C’s volatile attribute, and not for high performance synchronization between processors.

L32AI is used to load a synchronization variable. This load will be performed before any subsequent load, store, acquire, or release is begun. This ensures that subsequent loads and stores do not see or modify data that is protected by the synchronization variable.

S32RI is used to store to a synchronization variable. This store will not begin until all previous loads, stores, acquires, or releases are performed. This ensures that any loads of the synchronization variable that see the new value will also find all protected data available as well.

Consider the following example:

```
volatile uint incount = 0;
volatile uint outcount = 0;
const uint bsize = 8;
data_t buffer[bsize];
void producer (uint n)
{
        for (uint i = 0; i < n; i += 1) {
             data_t d = newdata();             // produce next datum
             while (outcount == i - bsize);    // wait for room
             buffer[i % bsize] = d;            // put data in buffer
             incount = i+1;                    // signal data is ready
        }
   }
   void consumer (uint n)
   {
        for (uint i = 0; i < n; i += 1) {
             while (incount == i);             // wait for data
             data_t d = buffer[i % bsize];     // read next datum
             outcount = i+1;                   // signal data read
             usedata (d);                      // use datum
        }
   }
```

Here, `incount` and `outcount` are synchronization variables, and `buffer` is a shared data variable. `producer`'s writes to `incount` and `consumer`'s writes to `outcount` must use `S32RI` and `producer`'s reads of `outcount` and `consumer`'s reads of incount must use `L32AI`. If `producer`'s write to `incount` were done with a simple `S32I`, the processor or memory system might reorder the write to `buffer` after the write to `incount`, thereby allowing `consumer` to see the wrong data. Similarly, if `consumer`'s read of incount were done with a simple `L32I`, the processor or memory system might reorder the read to `buffer` before the read of `incount`, also causing `consumer` to see the wrong data.

4.3.13 Conditional Store Option

In addition to the memory ordering needs satisfied by the Multiprocessor Synchronization Option, a multiprocessor system can require mutual exclusion, which cannot easily be programmed using the Multiprocessor Synchronization Option. The Conditional Store Option is intended to add that capability. It does so by adding a single instruction (`S32C1I`), which atomically stores to a memory location only if its current value is the expected one. A state register (`SCOMPARE1`) is also added to provide the additional operand required. Some implementations also have a state register (`ATOMCTL`) for further control of the atomic operation in cache and on the PIF bus.

- Prerequisites: Multiprocessor Synchronization Option (page 74)
- Incompatible Options: None

When the atomic operation reaches the PIF bus, it causes a Read-Compare-Write (RCW) transaction on the PIF, which is different from normal reads and writes.

4.3.13.1 Conditional Store Option Architectural Additions

Table 4–52 through Table 4–53 show this option's architectural additions.

Table 4–52. Conditional Store Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
SCOMPARE1	1	32	Conditional store comparison data	R/W	12
ATOMCTL[2]	1	6	Atomic Operation Control	R/W	99

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 3–23 on page 46.

[2] Register exists only in some implementations.

Table 4–53. Conditional Store Option Instruction Additions

Instruction[1]	Format	Definition
S32C1I	RRI8	Store 32-Bit compare conditional Stores to a location only if the location contains the value in the SCOMPARE1 register. The comparison of the old value and the store, if equal, is atomic. The instruction also returns the old value of the memory location.

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.3.13.2 Exclusive Access with the S32C1I Instruction

L32AI and S32RI allow inter-processor communication, as in the producer-consumer example in Section 4.3.12.3 (barrier synchronization is another example), but they are not efficient for guaranteeing exclusive access to data (for example, locks). Some systems may provide efficient, tailored, application-specific exclusion support. When this is not appropriate, the ISA provides another general-purpose mechanism for atomic updates of memory-based synchronization variables that can be used for exclusion algorithms. The S32C1I instruction stores to a location if the location contains the value in the SCOMPARE1 register. The comparison of the old value and the conditional store are atomic. S32C1I also returns the old value of the memory location, so it looks like both a load and a store; this allows the program to determine whether the store succeeded, and if not it can use the new value as the comparison for the next S32C1I. For example, an atomic increment could be done as follows:

```
        l32ai      a3, a2, 0            // current value of memory
    loop:
        wsr        a3, scompare1        // put current value in SCOMPARE1
        mov        a4, a3               // save for comparison
        addi       a3, a3, 1            // increment value
        s32c1i     a3, a2, 0            // store new value if memory
                                        // still contains SCOMPARE1
        bne        a3, a4, loop         // if value changed, try again
```

Semaphores and other exclusion operations are equally simple to create using S32C1I.

There are many possible atomic memory primitives. S32C1I was chosen for the Xtensa ISA because it can easily synthesize all other primitives that operate on a single memory location. Many other primitives (for example, test and set, or fetch and add) are not as universal. Only primitives that operate on multiple memory locations are more powerful than S32C1I. Note that there can be subtle issues with some algorithms if between a read and an S32C1I, there are multiple changes to the target which bring the value back to the original one.

The SCOMPARE1 register is undefined after reset.

4.3.13.3 Use Models for the S32C1I Instruction

Because of its nature as an atomic read-compare-write instruction, the S32C1I instruction is unusual in its relationships to local memories, caches, and system memories. Following is a list of ways that the S32C1I instruction is able to interact with memory. Some implementations use the ATOMCTL Special Register described below to control which way the instruction interacts with each memory type. Other implementations interact in a fixed way with each memory type. Refer to a specific Xtensa processor data book for more detailed information on how a specific processor handles S32C1I instructions.

- **Local Memory** — Xtensa processors with the Conditional Store Option and the Data RAM Option configured will execute S32C1I instructions whose address resolves to a DataRAM address directly on that DataRAM. Unless access to the DataRAM is shared with another master, no external logic is necessary in this case. None of the other ways listed below may be used for addresses resolving to a DataRAM.
- **Exception** — Xtensa processors with the Conditional Store Option and the Exception Option configured can execute the S32C1I instruction by taking an exception (LoadStoreErrorCause). The exception may be considered an error, or it may be used as a way to emulate the effect of the S32C1I instruction. Exception may be the only method available for certain memory types or it may be directed by the ATOMCTL register.
- **RCW Transaction** — Xtensa processors with the Conditional Store Option and the Processor Interface Option configured can execute the S32C1I instruction by sending an RCW transaction on the PIF bus. External logic must then implement the atomic read-compare-write on the memory location. If the Data Cache Option is configured and the memory region is cacheable, any corresponding cache line will be flushed out of the cache by the S32C1I instruction using the equivalent of a DHWBI instruction before the RCW transaction is sent. RCW Transaction may be the only method available for certain memory types or it may be directed by the ATOMCTL register.

If the address of the RCW transaction targets the Inbound PIF port of another Xtensa processor, the targeted Xtensa processor has the Conditional Store Option and the Data RAM Option configured, and the RCW address targets the DataRAM, the RCW will be performed atomically on the target processor's DataRAM. No external logic other than PIF bus interconnects is necessary to allow an Xtensa processor to atomically access a DataRAM location in another Xtensa processor in this way.

- **Internal Operation** — Xtensa processors with the Conditional Store Option and the Data Cache Option configured can execute the `S32C1I` instruction by allocating and filling the line in the cache and accessing the location atomically there. No external logic is necessary in this case. Internal Operation may be the only method available for certain memory types or it may be directed by the `ATOMCTL` register.

4.3.13.4 The Atomic Operation Control Register (ATOMCTL) under the Conditional Store Option

The `ATOMCTL` register exists in some implementations of the Conditional Store Option to control how the `S32C1I` instruction interacts with the cache and with the PIF bus. Implementations without the `ATOMCTL` register allow only one behavior per memory type. Table 4–54 shows the `ATOMCTL` register. Table 4–54 describes the fields of the `ATOMCTL` register. See Section 4.3.13.4 above for the meaning of the codes in the table.

31 6	5 4	3 2	1 0
reserved	WB	WT	BY
24	2	2	2

Table 4–54. ATOMCTL Register Fields

Field	Width (bits)	Definition
WB	2	S32C1I to Writeback Cacheable Memory (including Writeback NoAllocate Memory) 0 → Exception - LoadStoreErrorCause 1 → RCW Transaction 2 → Internal Operation 3 → Reserved
WT	2	S32C1I to Writethrough Cacheable Memory (including Cached-NoAllocate Memory) 0 → Exception - LoadStoreErrorCause 1 → RCW Transaction 2 → Internal Operation[1] 3 → Reserved
BY	2	S32C1I to Bypass Memory 0 → Exception - LoadStoreErrorCause 1 → RCW Transaction 2 → Reserved 3 → Reserved

[1] Some implementations do not implement this case and take an exception (LoadStoreErrorCause) instead.

ATOMCTL is defined after processor reset as shown in Table 5–186 on page 237.

An older, fixed operation, Xtensa processor which operates on all cacheable and bypass regions by RCW transaction may be emulated by setting the ATOMCTL register to 0x15. One which operates only on bypass regions by RCW transaction may be emulated by setting the ATOMCTL register to 0x01.

Bits of the ATOMCTL register are present even when they correspond to a memory type which is not configured in the Xtensa processor. For example, a processor configured without a Data Cache will still contain the fields WB and WT and those fields may contain any value. But in this case, no cacheable memory will be addressable and so it will not be possible to make use of these fields.

In an Xtensa processor with the Data RAM Option configured, the ATOMCTL register does not affect the "Local Memory" use model or the receiving of Inbound PIF transactions as described under the "RCW Transaction" use model in Section 4.3.13.3.

4.3.13.5 Memory Ordering and the S32C1I Instruction

With regard to the memory ordering defined for L32AI and S32RI in Section 4.3.12.1, S32C1I plays the role of both acquire and release. That is, before the atomic pair of memory accesses can perform, all ordinary loads, stores, acquires, and releases must have performed. In addition, before any following ordinary load, store, acquire, or release can be allowed to perform, the atomic pair of the S32C1I must have performed. This allows the conditional store to make atomic changes to variables with ordering requirements, such as the counts discussed in the example in Section 4.3.12.3.

4.4 Options for Interrupts and Exceptions

The options in this section have the primary function of adding and controlling the behavior of the processor in the presence of exceptional conditions. These conditions include representatives of at least the following broad categories:

- Instruction **exceptions** are unusual situations or errors encountered in the execution of the current instruction stream.
- **Interrupts** are requests from outside the instruction stream that, if enabled, can start the processor executing a different instruction stream.
- **Machine checks** are failures of the processor hardware or related hardware that need special handling to avoid causing the overall system to fail.
- **Debug** conditions do not arise from the execution of the program or the surrounding hardware, but rather from the desire of another agent to track the execution of the processor.
- **Reset** redirects the processor from any state, usually the undefined state after power-on, and starts it on a known execution path.

There are many ways of handling these conditions ranging from ignoring the conditions or freezing the clock and asserting an output signal to multi-threaded self-handling of exceptional conditions. The Exception Option provides for the self-handling of instruction exceptions and reset. Its self-handling mechanisms for these can be extended by the Relocatable Vector Option and the Unaligned Exception Option. In addition, it provides a foundation for additional options such as the Interrupt Option, the High-Priority Interrupt Option, or the Timer Interrupt Option. Again, the Debug Option can be added to provide for hardware debugging.

4.4.1 Exception Option

The Exception Option implements basic functions needed in the management of all types of exceptional conditions. These conditions are handled by the processor itself by redirecting execution to an exception vector to handle the condition with the possibility of returning to continue execution at the original code stream. The option only fully implements the management of a subset of exceptional conditions. Additional options providing additional exception types use the Exception Option as a foundation.

- Prerequisites: None
- Incompatible options: None

- Compatibility Note: Currently available hardware supports Xtensa Exception Architecture 2 (XEA2) and the descriptions in this chapter cover only XEA2. Differences between this and Xtensa Exception Architecture 1 (XEA1) are described, for purposes of writing system software for XEA1 processors, in Section A.2 on page 611.

4.4.1.1 Exception Option Architectural Additions

Table 4–55 through Table 4–58 show this option's architectural additions.

Table 4–55. Exception Option Constant Additions (Exception Causes)

Exception Cause	Constant Value
`IllegalInstructionCause`	`6'b000000` (decimal 0)
`SyscallCause`	`6'b000001` (decimal 1)
`InstructionFetchErrorCause`	`6'b000010` (decimal 2)
`LoadStoreErrorCause`	`6'b000011` (decimal 3)

Table 4–56. Exception Option Processor-Configuration Additions

Parameter	Description	Valid Values
`NDEPC`	Existence (number) of DEPC	0..1
`ResetVector`	Reset exception vector (PC of first instruction executed after reset)	32-bit address
`UserExceptionVector`	Vector for exceptions and level-1 interrupts when `PS.EXCM = 0` and `PS.UM = 1`	32-bit address
`KernelExceptionVector`	Vector for exceptions and level-1 interrupts when `PS.EXCM = 0` and `PS.UM = 0`	32-bit address
`DoubleExceptionVector`	Vector for exceptions when `PS.EXCM = 1`	32-bit address

Table 4–57. Exception Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
EPC[1]	1	32	Exception program counter[2]	R/W	177
EXCCAUSE	1	6	Cause of last exception[3]	R/W	232
EXCSAVE[1]	1	32	Save location for last exception[2]	R/W	209
PS	1	-[4]	Miscellaneous processor state[5]	R/W	230
PS.EXCM	1	4	Exception mode (see Table 4–63 on page 87)	R/W	230
PS.UM	1	1	User vector mode (see Table 4–63 on page 87)	R/W	230
EXCVADDR	1	32	Virtual address that caused last fetch, load, or store exception	R/W	238
DEPC	1	32	Double exception PC (exists if NDEPC=1)	R/W	192

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 3–23 on page 46.

[2] The EPC[i] and EXCSAVE[i] registers for interrupts above level 1 are part of the High-Priority Interrupt Option (Table 4–75 on page 107).

[3] See Table 4–64 on page 89 for the format of this register and Table 4–65 on page 94 for which vectors have causes reported in this register.

[4] Width depends on other configuration options.

[5] See "The Miscellaneous Program State Register (PS) under the Exception Option" on page 87.

Table 4–58. Exception Option Instruction Additions

Instruction[1]	Format	Definition
EXCW	RRR	Exception wait Waits for any exceptions of previously executed instructions to occur.
SYSCALL	RRR	System call Generates an exception.
RFE	RRR	Returns from the KernelExceptionVector exception.
RFDE	RRR	Returns from double exception (uses EPC if NDEPC=0)
ILL or illegal instruction	—	Illegal instruction executed The opcode ILL is guaranteed to always be an illegal instruction

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.4.1.2 Exception Causes under the Exception Option

A broad set of interrupts and exceptions can be handled by the processor itself under the Exception Option. Table 4–59 through Table 4–62 list the types of exceptional conditions other than reset that can be handled under the Exception Option either natively or with the help of an additional option. In each table, the first column contains the name of the condition. The second column contains a description of the condition and the third column contains both the option required for the condition to be handled and the name of the vector to which execution will be redirected. Reset is provided by the Exception Option and redirects execution to `ResetVector`.

Table 4–59. Instruction Exceptions under the Exception Option

Condition	Description	Required Option & Vector
Illegal instruction	Attempt to execute an illegal instruction or a legal instruction under illegal conditions	Exception Option General vector[1]
System call	Attempt to execute the `SYSCALL` instruction	Exception Option General vector[1]
Instruction fetch error	Internal physical address or a data error during instruction fetch	Exception Option General vector[1]
Load or store error	Internal physical address or data error during load or store	Exception Option General vector[1]
Unaligned data exception	Attempt to load or store data at an address which cannot be handled due to alignment	Unaligned Exception Option General vector[1]
Privileged instruction	Attempt to execute a privileged operation without sufficient privilege	MMU Option General vector[1]
Memory access prohibited	Attempt to access data or instructions at a prohibited address	Region Protection Option or MMU Option — General vector[1]
Memory privilege violation	Attempt to access data or instructions without sufficient privilege	MMU Option General vector[1]
Address translation failure	Memory access needs translation information it does not have available	MMU Option General vector[1]
PIF bus error	Address or data error external to the processor on the PIF bus[2]	Processor Interface Option General vector[1]

1. General Vector means.`DoubleExceptionVector` if `PS.EXCM` is set. Otherwise it means `UserExceptionVector` if `PS.UM` is set or `KernelExceptionVector` if `PS.UM` is clear.
2. Imprecise errors on writes are not included.
3. *n* can take on the values 4, 8, or 12 in each of overflow and underflow making a total of 6 vectors.

Table 4–59. Instruction Exceptions under the Exception Option (continued)

Condition	Description	Required Option & Vector
Window exception	Attempt to execute an instruction needing `AR` values moved between registers and stack	Windowed Register Option `WindowOverflow`*n*[3], or `WindowUnderflow`*n*[3]
Alloca exception	Attempt to move the stack pointer when it would cause an illegal condition on the stack	Windowed Register Option General vector[1]
Coprocessor disabled	Attempt to execute an instruction requiring the state of a disabled coprocessor	Coprocessor Option General vector[1]

[1] General Vector means.`DoubleExceptionVector` if `PS.EXCM` is set. Otherwise it means `UserExceptionVector` if `PS.UM` is set or `KernelExceptionVector` if `PS.UM` is clear.

[2] Imprecise errors on writes are not included.

[3] *n* can take on the values 4, 8, or 12 in each of overflow and underflow making a total of 6 vectors.

Condition	Description	Required Option & Vector
Level-1 interrupt	Level or edge interrupt pin assertion handled as part of general vector with software check	Interrupt Option General vector[1]
Level-1 SW interrupt	Version of level-1 interrupt caused by software using `WSR.INTSET`	Interrupt Option General vector[1]
Medium-Level interrupt	Level/edge interrupt pin assertion handled with special interrupt level, masked on stack unusable	High-Priority Interrupt Option `InterruptVector[2..6]`[2]
Medium-Level SW interrupt	Version of medium level interrupt caused by software using `WSR.INTSET`	High-Priority Interrupt Option `InterruptVector[2..6]`[2]
High-Level interrupt	Level/edge interrupt pin assertion handled with special interrupt level, extra stack care needed	High-Priority Interrupt Option `InterruptVector[2..6]`[2]
High-level SW interrupt	Version of high level interrupt caused by software using `WSR.INTSET`	High-Priority Interrupt Option `InterruptVector[2..6]`[2]
Non-maskable interrupt	Edge triggered interrupt pin that cannot be masked by software	High-Priority Interrupt Option `InterruptVector[2..7]`[2]
Peripheral interrupt	Internal hardware (e.g., timers) causes one of the above interrupts without an external pin	Timer Interrupt Option (asserts another interrupt type)

[1] General vector means.`DoubleExceptionVector` if `PS.EXCM` is set. Otherwise it means `UserExceptionVector` if `PS.UM` is set or `KernelExceptionVector` if `PS.UM` is clear.

[2] Medium and high level interrupts may use levels any level 2..6 not used for debug conditions. NMI is one level higher than the highest medium, high, or debug level.

Table 4–61. Machine Checks under the Exception Option

Condition	Description	Required Option & Vector
ECC/parity error	An access to cache or local memory produced an ECC or parity error	Memory ECC/Parity Option `MemoryErrorVector`

Table 4–62. Debug Conditions under the Exception Option

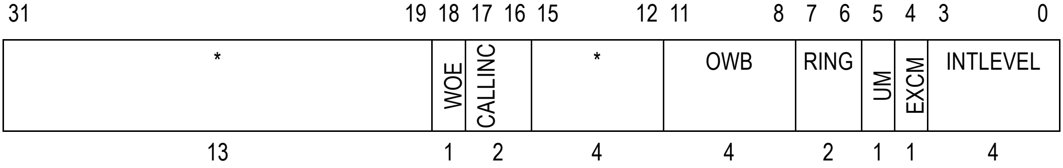
Condition	Description	Required Option & Vector
`ICOUNT` exception	An instruction would have incremented the `ICOUNT` register to zero.	Debug Option `InterruptVector[dbg]`[1]
`BREAK` exception	Attempt to execute the `BREAK` or `BREAK.N` instruction.	Debug Option `InterruptVector[dbg]`[1]
Instruction breakpoint	Attempt to execute an instruction matching one of the instruction breakpoint registers	Debug Option `InterruptVector[dbg]`[1]
Data breakpoint	Attempt to load or store to a data location matching one of the data breakpoint registers.	Debug Option `InterruptVector[dbg]`[1]
Debug interrupt	An interrupt through OCD	Debug Option[2] `InterruptVector[dbg]`[1]

[1] Debug exceptions use an interrupt level provided by the High-Priority Interrupt Option. That level is labeled "dbg" in this table.

[2] The debug interrupt is actually created by the OCD Option under the Debug Option.

4.4.1.3 The Miscellaneous Program State Register (`PS`) under the Exception Option

The `PS` register contains miscellaneous fields that are grouped together primarily so that they can be saved and restored easily for interrupts and context switching. Figure 4–8 shows its layout and Table 4–63 describes its fields. Section 5.3.5 “Processor Status Special Register” describes the fields of this register in greater detail. The processor initializes these fields on processor reset: `PS.INTLEVEL` is set to 15, if it exists and `PS.EXCM` is set to `1`, and the other fields are set to zero.



31 … 19	18	17 16	15 … 12	11 … 8	7 6	5	4	3 … 0
*	WOE	CALLINC	*	OWB	RING	UM	EXCM	INTLEVEL
13	1	2	4	4	2	1	1	4

Figure 4–8. PS Register Format

Table 4–63. PS Register Fields

Field	Width (bits)	Definition [Required Option]
`INTLEVEL`	4	Interrupt-level disable [Interrupt Option] Used to compute the current interrupt level of the processor (Section 4.4.1.4).
`EXCM`	1	Exception mode [Exception Option] 0 → normal operation 1 → exception mode Overrides the values of certain other PS fields (Section 4.4.1.4)

Table 4–63. PS Register Fields (continued)

Field	Width (bits)	Definition [Required Option]
UM	1	User vector mode [Exception Option] 0 → kernel vector mode — exceptions do not need to switch stacks 1 → user vector mode — exceptions need to switch stacks This bit does not affect protection. It is modified by software and affects the vector used for a general exception.
RING	2	Privilege level [MMU Option]
OWB	4	Old window base [Windowed Register Option] The value of WindowBase before window overflow or underflow.
CALLINC	2	Call increment [Windowed Register Option] Set to window increment by CALL instructions. Used by ENTRY to rotate window.
WOE	1	Window overflow-detection enable [Windowed Register Option] 0 → overflow detection disabled 1 → overflow detection enabled Used to compute the current window overflow enable (Section 4.4.1.4)
*		Reserved for future use. Writing a non-zero value to these fields results in undefined processor behavior.

4.4.1.4 Value of Variables under the Exception Option

The fields of the PS register listed in Table 4–63 affect many functions in the processor through these variables:

The current interrupt level (CINTLEVEL) defines which levels of interrupts are currently enabled and which are not. Interrupts at levels above CINTLEVEL are enabled. Those at or below CINTLEVEL are disabled. To enable a given interrupt, CINTLEVEL must be less than its level, and its INTENABLE bit must be 1. The level is defined by:

CINTLEVEL ← max(PS.EXCM*EXCMLEVEL,PS.INTLEVEL)

PS.EXCM and PS.INTLEVEL are part of the PS register in Table 4–63. EXCMLEVEL is defined in Table 4–74. CINTLEVEL is also used by the Debug Option.

The current ring (CRING) determines which ASIDs from the RASID register will cause a privilege violation. ASIDs with position (in RASID) equal to or greater than CRING may be used in translation while those with position less than CRING will cause a privilege violation. Privileged instructions may only be executed if CRING is zero. CRING is defined by:

CRING ← if (MMU Option configured && PS.EXCM = 0) then PS.RING else 0

PS.EXCM and PS.RING are part of the PS register in Table 4–63.

The current window overflow enable (CWOE) defines whether window overflow exceptions are currently enabled. It is defined by:

CWOE ← if PS.EXCM then 0 else PS.WOE

PS.EXCM and PS.WOE are part of the PS register in Table 4–63.

The current loop enable (CLOOPENABLE) determines whether the loop-back function of the zero-overhead loop instruction is enabled or not.

CLOOPENABLE ← PS.EXCM = 0

PS.EXCM is part of the PS register in Table 4–63.

4.4.1.5 The Exception Cause Register (EXCCAUSE) under the Exception Option

After an exception that redirects execution to one of the general exception vectors (UserExceptionVector, KernelExceptionVector, or DoubleExceptionVector), the EXCCAUSE register contains a value that specifies the cause of the last exception. Figure 4–9 shows the EXCCAUSE register. Table 4–64 describes the 6-bit binary-value encodings for the register. EXCCAUSE is undefined after processor reset.

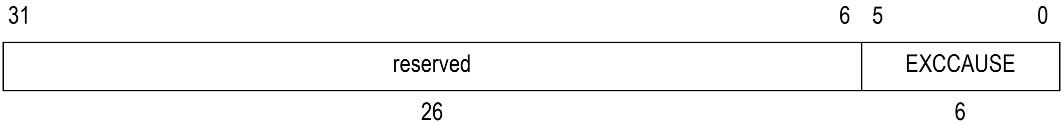


Figure 4–9. EXCCAUSE Register

Table 4–64. Exception Causes

EXC-CAUSE Code	Cause Name	Cause Description [Required Option]	EXC-VADDR Loaded
0	IllegalInstructionCause	Illegal instruction [Exception Option]	No
1	SyscallCause	SYSCALL instruction [Exception Option]	No
2	InstructionFetchErrorCause	Processor internal physical address or data error during instruction fetch [Exception Option]	Yes
3	LoadStoreErrorCause	Processor internal physical address or data error during load or store [Exception Option]	Yes
4	Level1InterruptCause	Level-1 interrupt as indicated by set level-1 bits in the INTERRUPT register [Interrupt Option]	No
5	AllocaCause	MOVSP instruction, if caller's registers are not in the register file [Windowed Register Option]	No

Table 4–64. Exception Causes (continued)

EXC-CAUSE Code	Cause Name	Cause Description [Required Option]	EXC-VADDR Loaded
6	IntegerDivideByZeroCause	QUOS, QUOU, REMS, or REMU divisor operand is zero [32-bit Integer Divide Option]	No
7		Reserved for Tensilica	
8	PrivilegedCause	Attempt to execute a privileged operation when CRING ≠ 0 [MMU Option]	No
9	LoadStoreAlignmentCause	Load or store to an unaligned address [Unaligned Exception Option]	Yes
10..11		Reserved for Tensilica	
12	InstrPIFDataErrorCause	PIF data error during instruction fetch [Processor Interface Option]	Yes
13	LoadStorePIFDataErrorCause	Synchronous PIF data error during LoadStore access [Processor Interface Option]	Yes
14	InstrPIFAddrErrorCause	PIF address error during instruction fetch [Processor Interface Option]	Yes
15	LoadStorePIFAddrErrorCause	Synchronous PIF address error during LoadStore access [Processor Interface Option]	Yes
16	InstTLBMissCause	Error during Instruction TLB refill [MMU Option]	Yes
17	InstTLBMultiHitCause	Multiple instruction TLB entries matched [MMU Option]	Yes
18	InstFetchPrivilegeCause	An instruction fetch referenced a virtual address at a ring level less than CRING [MMU Option]	Yes
19		Reserved for Tensilica	
20	InstFetchProhibitedCause	An instruction fetch referenced a page mapped with an attribute that does not permit instruction fetch [Region Protection Option or MMU Option]	Yes
21..23		Reserved for Tensilica	
24	LoadStoreTLBMissCause	Error during TLB refill for a load or store [MMU Option]	Yes
25	LoadStoreTLBMultiHitCause	Multiple TLB entries matched for a load or store [MMU Option]	Yes
26	LoadStorePrivilegeCause	A load or store referenced a virtual address at a ring level less than CRING [MMU Option]	Yes
27		Reserved for Tensilica	
28	LoadProhibitedCause	A load referenced a page mapped with an attribute that does not permit loads [Region Protection Option or MMU Option]	Yes

Table 4–64. Exception Causes (continued)

EXCCAUSE Code	Cause Name	Cause Description [Required Option]	EXCVADDR Loaded
29	StoreProhibitedCause	A store referenced a page mapped with an attribute that does not permit stores [Region Protection Option or MMU Option]	Yes
30..31		Reserved for Tensilica	
32..39	Coprocessor*n*Disabled	Coprocessor *n* instruction when cp*n* disabled. *n* varies 0..7 as the cause varies 32..39 [Coprocessor Option]	No
40..63		Reserved	

Exceptions that redirect execution to other vectors that do not use EXCCAUSE may either report details in a different cause register or may have only a single cause and no need for additional cause information.

4.4.1.6 The Exception Virtual Address Register (EXCVADDR) under the Exception Option

The exception virtual address (EXCVADDR) register contains the virtual byte address that caused the most recent fetch, load, or store exception. Table 4–64 shows, for every exception cause value, whether or not the exception virtual address register will be set. This register is undefined after processor reset. Because EXCVADDR may be changed by any TLB miss, even if the miss is handled entirely by processor hardware, code that counts on it not changing value must guarantee that no TLB miss is possible by using only static translations for both instruction and data accesses. Figure 4–10 shows the EXCVADDR register format.

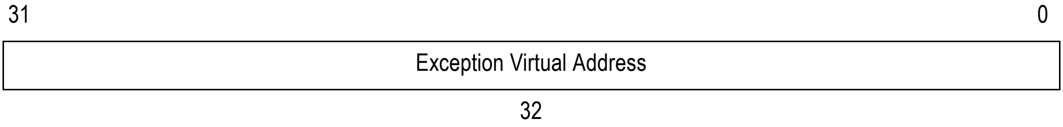


Figure 4–10. EXCVADDR Register Format

4.4.1.7 The Exception Program Counter (EPC) under the Exception Option

The exception program counter (EPC) register contains the virtual byte address of the instruction that caused the most recent exception or the next instruction to be executed in the case of a level-1 interrupt. This instruction has not been executed. Software may restart execution at this address by using the RFE instruction after fixing the cause of the exception or handling and clearing the interrupt. This register is undefined after processor reset and its value might change whenever PS.EXCM is 0.

The Exception Option defines only one EPC value (EPC[1]). The High-Priority Interrupt Option extends the EPC concept by adding one EPC value per high-priority interrupt level (EPC[2..NLEVEL+NNMI]).

Figure 4–11 shows the EPC register format.

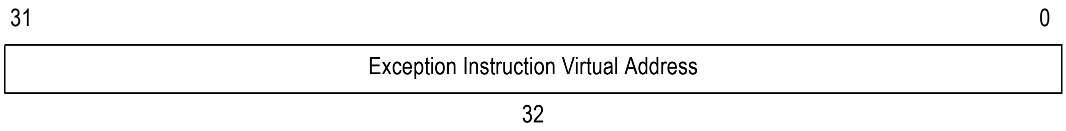


Figure 4–11. EPC Register Format for Exception Option

4.4.1.8 The Double Exception Program Counter (DEPC) under the Exception Option

The double exception program counter (DEPC) register contains the virtual byte address of the instruction that caused the most recent double exception. A double exception is one that is raised when PS.EXCM is set. This instruction has not been executed. Many double exceptions cannot be restarted, but those that can may be restarted at this address by using an RFDE instruction after fixing the cause of the exception.

The DEPC register exists only if the configuration parameter NDEPC=1. If DEPC does not exist, the EPC register is used in its place when a double exception is taken and when the RFDE instruction is executed. The consequence is that it is not possible to recover from most double exceptions. NDEPC=1 is required if both the Windowed Register Option and the MMU Option are configured. DEPC is undefined after processor reset.

Figure 4–12 shows the DEPC register format.

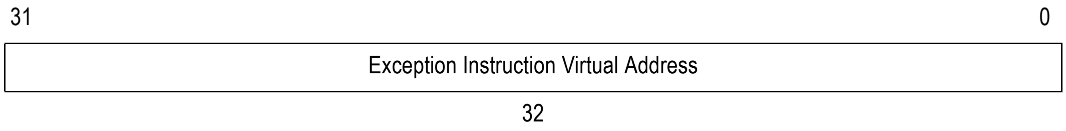


Figure 4–12. DEPC Register Format

4.4.1.9 The Exception Save Register (EXCSAVE) under the Exception Option

The exception save register (EXCSAVE[1]) is simply a read/write 32-bit register intended for saving one AR register in the exception vector software. This register is undefined after processor reset and there are many software reasons its value might change whenever PS.EXCM is 0.

The Exception Option defines only one exception save register (EXCSAVE[1]). The High-Priority Interrupt Option extends this concept by adding one EXCSAVE register per high-priority interrupt level (EXCSAVE[2..NLEVEL+NNMI]).

Figure 4–13 shows the `EXCSAVE` register format.

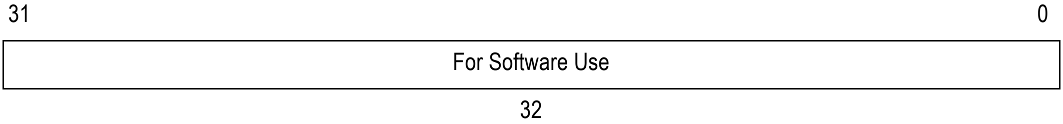


Figure 4–13. EXCSAVE Register Format

4.4.1.10 Handling of Exceptional Conditions under the Exception Option

Under the Exception Option, exceptional conditions are handled by saving some state and redirecting execution to one of a set of exception vector locations as listed in Table 4–59 through Table 4–62 along with `ResetVector`. This section looks at this process from the other end and describes how the code at a vector can determine the nature of the exceptional condition that has just occurred.

Table 4–65 shows, for each vector, how the code can determine what has happened. The first column lists the possible vectors, not just for the Exception Option itself, but also for other options that add on to the Exception Option. For vectors which can be reached for more than one cause, the second column indicates the register containing the main indicator of that cause. The third column indicates other registers that may contain secondary information under that vector. The last column shows the option that is required for the vector and the other listed registers to exist.

The three exception vectors that use `EXCCAUSE` for the primary cause information form a set called the “general vector.” If `PS.EXCM` is set when one of the exceptional conditions is raised, then the processor is already handling an exceptional condition and the exception goes to the `DoubleExceptionVector`. Only a few double exceptions are recoverable, including a TLB miss during a register window overflow or underflow exception. For these, `EXCCAUSE` (and `EXCSAVE` in Table 4–66) must be well enough understood not to need duplication. Otherwise (`PS.EXCM` clear), if `PS.UM` is set the exception goes to the `UserExceptionVector`, and if not the exception goes to the `KernelExceptionVector`. The Exception Option effectively defines two operating modes: user vector mode and kernel vector mode, controlled by the `PS.UM` bit. The combination of user vector mode and kernel vector mode is provided so that the user vector exception handler can switch to an exception stack before processing the exception, whereas the kernel vector exception handler can continue using the kernel stack.

Single or multiple high-priority interrupts can be configured for any hardware prioritized levels 2..6. These will redirect to the `InterruptVector[i]` where “`i`” is the level. One of those levels, often the highest one, can be chosen as the debug level and will redirect execution to `InterruptVector[d]` where “`d`” is the debug level. The level one higher than the highest high-priority interrupt can be chosen as an NMI, which will redirect execution to `InterruptVector[n]` where “`n`” is the NMI level (2..7).

Table 4–65. Exception and Interrupt Information Registers by Vector

Vector	Main Cause	Other Information	Required Option
ResetVector	—	—	Exception Option
UserExceptionVector	EXCCAUSE	INTERRUPT, EXCVADDR	Exception Option
KernelExceptionVector	EXCCAUSE	INTERRUPT, EXCVADDR	Exception Option
DoubleExceptionVector	EXCCAUSE	EXCVADDR	Exception Option
WindowOverflow4	—	—	Windowed Register Option
WindowOverflow8	—	—	Windowed Register Option
WindowOverflow12	—	—	Windowed Register Option
WindowUnderflow4	—	—	Windowed Register Option
WindowUnderflow8	—	—	Windowed Register Option
WindowUnderflow12	—	—	Windowed Register Option
MemoryErrorVector	MESR	MECR, MEVADDR	High-Priority Interrupt Option
InterruptVector[i][1]	INTERRUPT	—	High-Priority Interrupt Option
InterruptVector[d][2]	DEBUGCAUSE	—	Debug Option
InterruptVector[n][3]	—	—	High-Priority Interrupt Option

[1] "i" indicates an arbitrary interrupt level. Medium- and high-level interrupts may be levels 2..6.

[2] "d" indicates the debug level. It may be levels 2..6 but is usually the highest level other than NMI.

[3] "n" indicates the NMI level. It may be levels 2..7. It must be the highest level but contiguous with other levels.

In addition to these characteristics of Vectors, when the Relocatable Vector Option (page 98) is configured, the vectors are divided into two groups and within each group are required to be in increasing address order as listed below:

Static Vector Group:

- ResetVector
- MemoryErrorVector

Dynamic Vector Group:

- WindowOverflow4
- WindowUnderflow4
- WindowOverflow8
- WindowUnderflow8
- WindowOverflow12
- WindowUnderflow12
- InterruptVector[2]

- `InterruptVector[3]`
- `InterruptVector[4]`
- `InterruptVector[5]`
- `InterruptVector[6]`
- `InterruptVector[7]`
- `KernelExceptionVector`
- `UserExceptionVector`
- `DoubleExceptionVector`

Table 4–66 shows, for each vector in the first column, which registers are involved in the process of taking the exception and returning from it for that vector. Since there is no return from the `ResetVector`, it has no entries in the other four columns of this table. Otherwise all entries have a second column entry of where the PC is saved and a fifth column entry of the instruction which should be used for returning. The third column shows where the current PS register value is saved before being changed, while the fourth column shows where the handler may find a scratch register. Note that the general vector entries and the window vector entries modify the PS only in ways that their respective return instructions undo, and therefore there is no required PS save register. The window vector entries do not need scratch space because they are loading and storing a block of AR registers that they can use for scratch where they need it.

Table 4–66. Exception and Interrupt Exception Registers by Vector

Vector	PC	PS	Scratch	Return Instr.
ResetVector	—	—	—	—
UserExceptionVector	EPC	—	EXCSAVE	RFE
KernelExceptionVector	EPC	—	EXCSAVE	RFE
DoubleExceptionVector	DEPC	—	EXCSAVE	RFDE
WindowOverflow4	EPC	—	—	RFWO
WindowOverflow8	EPC	—	—	RFWO
WindowOverflow12	EPC	—	—	RFWO
WindowUnderflow4	EPC	—	—	RFWU
WindowUnderflow8	EPC	—	—	RFWU
WindowUnderflow12	EPC	—	—	RFWU
MemoryErrorVector	MEPC	MEPS	MESAVE	RFME

1. "`i`" indicates an arbitrary interrupt level. Medium- and high-level interrupts may be levels 2..6.
2 "`d`" indicates the debug level. It may be levels 2..6 but is usually the highest level other than NMI.
3. "`n`" indicates the NMI level. It may be levels 2..7. It must be the highest level but contiguous with other levels.

Table 4–66. Exception and Interrupt Exception Registers by Vector (continued)

Vector	PC	PS	Scratch	Return Instr.
`InterruptVector[i]`[1]	`EPCi`[1]	`EPSi`[1]	`EXCSAVEi`[1]	`RFIi`[1]
`InterruptVector[d]`[2]	`EPCd`[2]	`EPSd`[2]	`EXCSAVEd`[2]	`RFId`[2]
`InterruptVector[n]`[3]	`EPCn`[3]	`EPSn`[3]	`EXCSAVEn`[3]	`RFIn`[3]

[1] "`i`" indicates an arbitrary interrupt level. Medium- and high-level interrupts may be levels 2..6.

[2] "`d`" indicates the debug level. It may be levels 2..6 but is usually the highest level other than NMI.

[3] "`n`" indicates the NMI level. It may be levels 2..7. It must be the highest level but contiguous with other levels.

The taking of an exception under the Exception Option has the following semantics:

```
procedure Exception(cause)
     if (PS.EXCM & NDEPC=1) then
          DEPC ← PC
          nextPC ← DoubleExceptionVector
     elseif PS.EXCM then
          EPC[1] ← PC
          nextPC ← DoubleExceptionVector
     elseif PS.UM then
          EPC[1] ← PC
          nextPC ← UserExceptionVector
     else
          EPC[1] ← PC
          nextPC ← KernelExceptionVector
     endif
     EXCCAUSE ← cause
     PS.EXCM ← 1
endprocedure Exception
```

4.4.1.11 Exception Priority under the Exception Option

In implementations where instruction execution is overlapped (for example, via a pipeline), multiple instructions can cause exceptions. In this case, priority is given to the exception caused by the *earliest instruction*.

When a given instruction causes multiple exceptions, the priority order for choosing the exception to be reported is listed below from highest priority to lowest. In cases where it is possible to have more than one occurrence of the same cause within the same instruction, the priority among the occurrences is undefined.

Pre-Instruction Exceptions:

- Non-maskable interrupt
- High-priority interrupt (including debug exception for `DEBUG INTERRUPT)`
- Level1InterruptCause

- Debug exception for `ICOUNT`
- Debug exception for `IBREAK`

Fetch Exceptions:

- Instruction-fetch translation errors
  - InstTLBMultiHitCause
  - InstTLBMissCause
  - InstFetchPrivilegeCause
  - InstFetchProhibitedCause
- InstructionFetchErrorCause (Instruction-fetch address or instruction data errors)
- ECC/parity exception for Instruction-fetch

Decode Exceptions:

- IllegalInstructionCause
- PrivilegedCause
- SyscallCause (`SYSCALL` instruction)
- Debug exception for `BREAK` (`BREAK`, `BREAK.N` instructions)

Execute Register Exceptions:

- Register window overflow
- Register window underflow (`RETW`, `RETW.N` instructions)
- AllocaCause (`MOVSP` instruction)
- Coprocessor*n*DisabledCause

Execute Data Exceptions:

Divide by Zero

Execute Memory Exceptions:

- LoadStoreAlignmentCause (in the absence of the Hardware Alignment Option)
- Debug exception for `DBREAK`
- `IHI`, `PITLB`, `IPF`, or `IPFL`, or `IHU` target translation errors, in order of priority:
  - InstTLBMultiHitCause
  - InstTLBMissCause
  - InstFetchPrivilegeCause
  - InstFetchProhibitedCause

- Load, store, translation errors, in order of priority:
  - LoadStoreTLBMultiHitCause
  - LoadStoreTLBMissCause
  - LoadStorePrivilegeCause
  - LoadProhibitedCause
  - StoreProhibitedCause
- InstructionFetchErrorCause (`IPFL` target address or data errors)
- LoadStoreAlignmentCause (in the presence of the Hardware Alignment Option)
- LoadStoreErrorCause (Load or store external address or data errors)
- ECC/parity exception for all accesses except instruction-fetch

Exceptions are grouped in the priority list by what information is necessary to determine whether or not the exception is to be raised. The pre-instruction exceptions may be evaluated before the instruction begins because they require nothing but the PC of the instruction. Fetch exceptions are encountered in the process of fetching the instruction. Decode exceptions may be evaluated after obtaining the instruction itself. Execute register exceptions require internal register state and execute memory exceptions involve the process of accessing the memory on which the instruction operates.

Exceptions are not necessarily precise. On some implementations, some exceptions are raised after subsequent instructions have been executed. In such implementations, the `EXCW` instruction can be used to prevent unwanted effects of imprecise exceptions. The `EXCW` instruction causes the processor to wait until all previous instructions have taken their exceptions, if any.

Interrupts have an implicit `EXCW`; when an interrupt is taken, all instructions prior to the instruction addressed by `EPC` have been executed and any exceptions caused by those instructions have been raised. Interrupts are listed at the top of the priority list. Because the relative cycle position of an internal instruction and an interrupt pin assertion is not well-defined, the priority of interrupts with respect to exceptions is not truly well-defined either.

4.4.2 Relocatable Vector Option

This option splits Exception Vectors into two groups and adds a choice of two base addresses for one group and a Special Register as a base for the other group.

- Prerequisites: Exception Option (page 82)
- Incompatible options: None

Under the Relocatable Vector Option, exception vectors are more restricted than they are without it. The vectors are organized into two groups, a "Static" group and a "Dynamic" group. Within each group there is a required order among the vectors which exist. The list immediately after Table 4–65 (page 94) indicates both the group and the order within the group. Some implementations may place an upper bound on the size of each group of vectors as measured by the difference between the address of the highest numbered vector in the group and the address of the lowest numbered vector in the group.

The Static group of vectors is not movable under software control. Two base addresses for the Static group are set by the designer at configuration time and an input pin of the processor is sampled at reset to determine which of the two configured addresses will be used. The base address will not change after reset. The offsets from this base are also chosen at configuration time and will not change.

The Dynamic group of vectors is movable under software control. The Special Register, `VECBASE`, described in Table 5–155 on page 224, holds the current base for the Dynamic group. The special register resets to a value set by the designer at configuration time but is freely writable using the `WSR.VECBASE` instruction. The offsets from the base must increase in the order indicated by Table 4–66 and are also set by the designer at configuration time.

4.4.2.1 Relocatable Vector Option Architectural Additions

Table 4–67 shows this option’s architectural additions.

Table 4–67. Relocatable Vector Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
`VECBASE`	1	28	Vector base	R/W	Table 5–155

4.4.3 Unaligned Exception Option

This option causes an exception to be raised on any unaligned memory access whether it is generated by core architecture memory instructions, by optional instructions, or by a designer’s TIE instructions.[1] With system software cooperation, occasional unaligned accesses can be handled correctly.

[1] In the T1050 release, which was the first for the Unaligned Exception Option, only Core Architecture memory instructions raise the unaligned exception.

Cache line oriented instructions such as prefetch and cache management instructions will not raise the unaligned exception. Special instructions such as `LICW` that use a generated address for something other than an actual memory address also will not raise the exception. Individual instruction listings list the unaligned exception when it can be raised by that instruction.

Memory access instructions will raise the exception when address and size indicate it. Any address that is not a multiple of the size associated with the instruction will raise the unaligned exception whether or not the access crosses any particular size boundary. For example, an `L16UI` instruction that generates the address `32'h00000005`, will raise the unaligned exception, even though the access is entirely within a single 32-bit access.

The exception cause register will contain `LoadStoreAlignmentCause` as indicated below and the exception virtual address register will contain the virtual address of the unaligned access.

- Prerequisites: Exception Option (page 82)
- Incompatible options: None

4.4.3.1 Unaligned Exception Option Architectural Additions

Table 4–68 shows this option's architectural additions.

Table 4–68. Unaligned Exception Option Constant Additions (Exception Causes)

Exception Cause	Description	Constant Value
`LoadStoreAlignmentCause`	Load or store to an unaligned address. (seeTable 4–64 on page 89)	`6'b001001` (decimal 9)

4.4.4 Interrupt Option

The Interrupt Option implements level-1 interrupts. These are asynchronous exceptions on processor input signals or software exceptions. They have the lowest priority of all interrupts. Level-1 interrupts are handled differently than the high-priority interrupts at priority levels 2 through 6 or NMI. The Interrupt Option is a prerequisite for the High-Priority Interrupt Option, Timer Interrupt Option, and Debug Option.

Certain aspects of high-priority interrupts are specified along with those of level-1 interrupts in the Interrupt Option. Specifically, the following parameters are specified:

- `NINTERRUPT`—Total number of level-1 plus high-priority interrupts.
- `INTTYPE[0..NINTERRUPT-1]`—Interrupt type (level, edge, software, or internal) for level-1 plus high-priority interrupts.
- `INTENABLE`—Interrupt-enable mask for level-1 plus high-priority interrupts.

- INTERRUPT—Interrupt-request register for level-1 plus high-priority interrupts.

Nevertheless, high-priority interrupts specified in the Interrupt Option are not operational without implementation of the High-Priority Interrupt Option.

- Prerequisites: Exception Option (page 82)
- Incompatible options: None

4.4.4.1 Interrupt Option Architectural Additions

Table 4–69 through Table 4–72 show this option's architectural additions.

Table 4–69. Interrupt Option Constant Additions (Exception Causes)

Exception Cause	Description	Constant Value
Level1InterruptCause	Level-1 interrupt (seeTable 4–64 on page 89)	6'b000100 (decimal 4)

Table 4–70. Interrupt Option Processor-Configuration Additions

Parameter	Description	Valid Values
NINTERRUPT	Number of level-1, high-priority, and non-maskable interrupts	1..32
INTTYPE[0..NINTERRUPT-1]	Interrupt type for level-1, high-priority, and non-maskable interrupts Section 4.4.4.2	See Table 4–73
LEVEL[0..NINTERRUPT-1]	Priority level of level-1 interrupts[1]	1

[1] This parameter has a fixed, implicit value. The parameter associates the level-1 interrupts with their interrupt priority (level) which, by definition, is always level 1 (lowest priority), The parameter must be explicitly specified only for the high-priority interrupts (Table 4–74 on page 107), each of which can be assigned different priority levels, from 2 to 15.

Table 4–71. Interrupt Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
PS.INTLEVEL	1	4	Interrupt-level disable (see Table 4–63 on page 87)	R/W	See Table 4–63 on page 87

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 3–23 on page 46.

[2] Level-sensitive interrupt bits are read-only, edge-triggered interrupt bits are read/clear, and software interrupt bits are read/write. Two register numbers are provided for software modification to the INTERRUPT register: one that sets bits, and one that clears them.

Table 4–71. Interrupt Option Processor-State Additions (continued)

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
INTENABLE	1	NINTERRUPT	Interrupt enable mask (Level-1 and high-priority interrupts) There is one bit for each level-1 and high-priority interrupt, except non-maskable interrupt (NMI) and Debug interrupt. To enable a given interrupt, CINTLEVEL (Table 4–57 on page 84) must be less than the level assigned by LEVEL[i] to that interrupt, and the INTENABLE bit for that interrupt must be set to 1.	R/W	228
INTERRUPT (the mnemonics INTERRUPT, INTSET, and INTCLEAR are used depending on the type of access)	1	NINTERRUPT	Interrupt request register (level-1 and high-priority interrupts) This holds pending level-1 and high-priority interrupt requests. There is 1 bit per pending interrupt, except non-maskable interrupt (NMI). If the bit is set to 1, an interrupt request is pending. External level interrupt bits are not writable.	R or R/W[2]	226 for read, 226 for set, and 227 for clear

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 3–23 on page 46.

[2] Level-sensitive interrupt bits are read-only, edge-triggered interrupt bits are read/clear, and software interrupt bits are read/write. Two register numbers are provided for software modification to the INTERRUPT register: one that sets bits, and one that clears them.

Table 4–72. Interrupt Option Instruction Additions

Instruction[1]	Format	Definition
RSIL	RRR	Read and set interrupt level
WAITI	RRR	Wait for interrupt

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.4.4.2 Specifying Interrupts

Interrupt types (INTTYPE in Table 4–70) can be any of the values listed in Table 4–73. The column labeled “Priority” shows the possible range of priorities for the interrupt type. The column labeled “Pin” indicates whether there is an Xtensa core pin associated with the interrupt, while the column labeled “Bit” indicates whether or not there is a bit in the INTERRUPT and INTENABLE Special Registers corresponding to the interrupt. The last two columns indicate how the interrupt may be set and how it may be cleared.

Table 4–73. Interrupt Types

Type	Priority[1]	Pin?	Bit?	How Interrupt is Set	How Interrupt is Cleared
`Level`	1 to N	Yes	Yes	Signal level from device	At device
`Edge`	1 to N	Yes	Yes	Signal rising edge	`WSR.INTCLEAR ‘1’`
`NMI`	N+1	Yes	No	Signal rising edge	Automatically cleared by HW
`Software`	1 to N	No	Yes	`WSR.INTSET ‘1’`	`WSR.INTCLEAR ‘1’`
`Timer`	1 to N	No	Yes	`CCOUNT=CCOMPAREn`	`WSR.CCOMPAREn`
`Debug`[2]	2 to N	No[2]	No	Debug hardware[2]	Automatically cleared by HW
`WriteErr`	1 to N	No	Yes	Bus error on write	`WSR.INTCLEAR ‘1’`

[1] Possible priorities where N is NLEVEL

[2] SeeSection 4.7.6 “Debug Option” on page 197 for more detail

A variable number (`NINTERRUPT`) of interrupts can be defined during processor configuration. External interrupt requests are signaled to the processor by either level-sensitive or edge-triggered inputs. Software can test these interrupt requests at any time by reading the `INTERRUPT` register. An arbitrary number of software interrupts, not tied to an external signal, can also be configured. Level-1 interrupts use either the `UserExceptionVector` or `KernelExceptionVector` defined in Table 4–56 on page 83, depending on the current setting of the `PS.UM` bit.

Software can manipulate the interrupt-enable bits (`INTENABLE` register) and then set `PS.INTLEVEL` back to 0 to re-enable other interrupts, and thereby create arbitrary prioritizations. This is illustrated by the following C++ code:

```
class Interrupt {
public:
  uint32_t bit;
  void handler();
};

class Level1Interrupt {
  const uint NPRIORITY = 4;         // number of priority groupings of level1 interrupts
  struct InterruptGroup {
    uint32_t allbits;               // all INTERRUPT register bits at this priority
    uint32_t mask;                  // mask of interrupt bits at this priority and lower
    vector<Interrupt> intlist;      // list of interrupts at this priority
  } priority[NPRIORITY];
public:
  void handler();
};


// Called for all Level1 Interrupts
void
Level1Interrupt::handler ()
{
  // determine software priority of this level1 interrupt
  uint32_t interrupts = rsr(INTERRUPT);
  uint p;
  for (p = NPRIORITY-1; (interrupts & priority[p].allbits) == 0; p -= 1) {
    if (p == 0)
      return;
  }
  // found interrupts at priority p
  uint32_t save_enable = rsr(INTENABLE);// save interrupt enables
  wsr (INTENABLE, save_enable &~ priority[p].mask);// disable lower-priority ints
  // no xSYNC instruction should be necessary here because INTENABLE and
  // PS.INTLEVEL are both written and both used in the same pipe stages
  uint32_t save_ps = rsil (0);      // save PS, then set level to 0
  // now higher-priority level1 interrupts are enabled
  // service all the priority p interrupts
  do {
    // first service the priority p interrupts we read earlier
    for (vector<Interrupt>::iterator i = priority[p].intlist.begin();
          i = priority[p].intlist.end(); i++) {
      if (interrupts & i->bit) {
                 // interrupt i is asserted
                 i->handler();      // call i's handler
                                    // this should clear the interrupt condition before it returns
                 interrupts &= ~i->bit;// clear i's bit from request
                 if ((interrupts & priority[p].allbits) == 0)// early check for done
                   break;
      }
    }
    // check if any more priority p interrupts arrived while we were servicing the previous batch
    interrupts = rsr(INTERRUPT);
  } while ((interrupts & priority[p].allbits) == 0);
  // no more priority p interrupts
wsr (PS, save_ps);                 // return to PS.INTLEVEL=1, disabling
                                   // all level1 interrupts, before returning
wsr (INTENABLE, save_enable);      // restore original enables to allow lower
                                   // priority level1 interrupts
// return to general exception handler
}
```

4.4.4.3 The Level-1 Interrupt Process

With respect to level-1 interrupts, the processor takes an interrupt when any level-1 interrupt, `i`, satisfies:

$$\texttt{INTERRUPT}_i \text{ and } \texttt{INTENABLE}_i \text{ and } (1 > \texttt{CINTLEVEL})$$

Level-1 interrupts use the `UserExceptionVector` and `KernelExceptionVector`, implemented by the Exception Option (Table 4–56 on page 83). The interrupt cause is reported as `Level1InterruptCause` (Table 4–64). The interrupt handler can determine which level-1 interrupt caused the exception by doing an `RSR` of the `INTERRUPT` register and ANDing with the contents of the `INTENABLE` register. The exact semantics of the check for interrupts is given in "Checking for Interrupts" on page 109.

The process of taking an interrupt does not clear the interrupt request. The process does set `PS.EXCM` to 1, which disables level-1 interrupts in the interrupt handler. Typically, `PS.EXCM` is reset to 0 by the handler, after it has set up the stack frame and masked the interrupt. This allows other level-1 interrupts to be serviced. For level-sensitive interrupts, the handler must cause the source of the interrupt to deassert its interrupt request before re-enabling the interrupt. For edge-triggered interrupts or software interrupts, the handler clears the interrupt condition by writing to the `INTCLEAR` register.

The `WAITI` instruction sets the current interrupt level in the `PS.INTLEVEL` register. In some implementations it also powers down the processor's logic, and waits for an interrupt. After executing the interrupt handler, execution continues with the instruction following the `WAITI`.

The `INTENABLE` register and the software and edge-triggered bits of the `INTERRUPT` register are undefined after processor reset.

4.4.4.4 Use of Interrupt Instructions

The RSIL instruction reads the PS register and sets the interrupt level. It is typically used as follows:

```
RSIL      a2, newlevel
code to be executed at newlevel
WSR       a2, PS
```

A SYNC instruction is not required after the RSIL.

4.4.5 High-Priority Interrupt Option

The High-Priority Interrupt Option implements a configurable number of interrupt levels between level 2 and level 6, and an optional non-maskable interrupt (NMI) at an implicit infinite priority level. Like level-1 interrupts, high-priority interrupts are external, internal or software interrupts. Unlike level-1 interrupts, however, each high-priority interrupt level has its own interrupt vector and special registers dedicated for saving state (EPC[level], EPS[level] and EXCSAVE[level]). This allows much lower latency interrupts as well as very efficient handler mechanisms. The EPC, EPS and EXCSAVE registers are undefined after reset.

Certain aspects of high-priority interrupts are specified along with those of level-1 interrupts in the Interrupt Option, including the total number of level-1 plus high-priority interrupts (NINTERRUPT), the interrupt type for level-1 plus high-priority interrupts (INTTYPE), the interrupt-enable mask for level-1 plus high-priority interrupts (INTENABLE), and the interrupt-request register for level-1 plus high-priority interrupts (INTERRUPT).

- Prerequisites: Interrupt Option (page 100)
- Incompatible options: None

4.4.5.1 High-Priority Interrupt Option Architectural Additions

Table 4–74 through Table 4–76 show this option’s architectural additions.

Table 4–74. High-Priority Interrupt Option Processor-Configuration Additions

Parameter	Description	Valid Values
NLEVEL	Number of high-priority interrupt levels	2..6[1]
EXCMLEVEL	Highest level masked by PS.EXCM	1..NLEVEL[2]
NNMI	Number of non-maskable interrupts (NMI)	0 or 1
LEVEL[0..NINTERRUPT-1]	Priority levels of interrupts	1..NLEVEL[3]
InterruptVector[2..NLEVEL+NNMI]	High-priority interrupt vectors	32-bit address, aligned on a 4-byte boundary
LEVELMASK[1..NLEVEL-1]	Interrupt-level masks	computed[4]

[1] An interrupt's "level" expresses its priority. The NLEVEL parameter defines the number of total interrupt levels (including level 1). Without the High-Priority Interrupt Option, NLEVEL is fixed at 1. With the High-Priority Interrupt Option, NLEVEL ≥ 2.

[2] EXCMLEVEL was required to be 1 before the RA-2004.1 release. In the presence of the Debug Option, it still must be less than DEBUGLEVEL.

[3] This parameter associates interrupt levels (priorities) with interrupt numbers. level-1 interrupts, by definition, are always priority level 1 (lowest priority), and are defined in Table 4–70 on page 101. Non-maskable interrupts (NMI) have many characteristics of the level NLEVEL+1. There is no level 0.

[4] This is computed as: LEVELMASK[j]i = (LEVEL[i] = j), where j is the level specified for interrupt i, and the width of each LEVELMASK is NINTERRUPT. Thus, there are NLEVEL-1 masks (one for each high-priority interrupt level), and each mask is NINTERRUPT bits wide. A bit number set to 1 in a LEVELMASK means that the corresponding interrupt number has that priority level. The masks are used in the formal semantics to test whether an interrupt is taken on a given instruction ("Checking for Interrupts" on page 109).

Table 4–75. High-Priority Interrupt Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
EPC [2..NLEVEL+NNMI]	NLEVEL+NNMI-1	32	Exception program counter	R/W	178-183
EPS [2..NLEVEL+NNMI]	NLEVEL+NNMI-1	same as PS register	Exception program state	R/W	194-199
EXCSAVE [2..NLEVEL+NNMI]	NLEVEL+NNMI-1	32	Save Location for high-priority interrupt handler	R/W	210-215

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 3–23 on page 46.

Table 4–76. High-Priority Interrupt Option Instruction Additions

Instruction[1]	Format	Definition
RFI	RRR	Return from high-priority interrupt

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.4.5.2 Specifying High-Priority Interrupts

The total number of level-1 plus high-priority interrupts (NINTERRUPT) and the interrupt type for level-1 plus high-priority interrupts (INTTYPE) are specified in Table 4–70 on page 101. The type of each high-priority interrupt level may be edge-triggered, level-sensitive, timer, write-error, or software.

The interrupt-enable mask for level-1 plus high-priority interrupts (INTENABLE) and the interrupt-request register for level-1 plus high-priority interrupts (INTERRUPT) are specified in Table 4–71 on page 101.

The total number of interrupt levels is NLEVEL+NNMI (see Table 4–74). Specific interrupt numbers are assigned interrupt levels using the LEVEL parameter in Table 4–74. A non-maskable interrupt may be configured with the NNMI parameter in Table 4–74. The non-maskable interrupt signal, if implemented, will be edge-triggered. Unlike other edge-triggered interrupts, there is no need to reset the NMI interrupt by writing to INTCLEAR.

4.4.5.3 The High-Priority Interrupt Process

Each high-priority interrupt level has three registers used to save processor state, as shown in Table 4–75. The processor sets EPC[i] and EPS[i] when the interrupt is taken. EXCSAVE[i] exists for software. The RFI instruction reverses the interrupt process, restoring processor state from EPC[i] and EPS[i].

The number of high-priority interrupt levels is expected to be small, due to the cost of providing separate exception-state registers for each level. Interrupt numbers that share level 1 are not limited to a single priority, because software can manipulate the interrupt-enables bits (INTENABLE register) to create arbitrary prioritizations.

The processor takes an interrupt only when some interrupt i satisfies:

$$\text{INTERRUPT}_i \text{ and } \text{INTENABLE}_i \text{ and } (\text{level}[i] > \text{CINTLEVEL})$$

where level[i] is the configured interrupt level of interrupt number i. Each level of high-priority interrupt has its own interrupt vector (InterruptVector in Table 4–74). Interrupt numbers that share a level (and associated vector) can read the INTERRUPT register (and INTENABLE) with the RSR instruction to determine which interrupt(s) raised the exception. The non-maskable interrupt (NMI), if implemented, is taken regardless of the current interrupt level (CINTLEVEL) or of INTENABLE.

The value of CINTLEVEL is set to at least EXCMLEVEL whenever PS.EXCM=1. Thus, all interrupts at level EXCMLEVEL and below are masked during the time PS.EXCM=1. This is done to allow high-level language coding with the Windowed Register Option of interrupt handlers for interrupts whose level is not greater than EXCMLEVEL. High-priority interrupts with levels at or below EXCMLEVEL are often called medium-priority interrupts. The interrupt latency is somewhat lower for levels greater than EXCMLEVEL, but handlers are more flexible for those whose level is not greater than EXCMLEVEL.

There are other conditions besides those in this section that can postpone the taking of an interrupt. For more descriptions on these, refer to a specific Xtensa processor data book.

4.4.5.4 Checking for Interrupts

The example below checks for interrupts. This is the checkInterrupts() procedure called in the code example shown in Section 3.5.4 “Instruction Fetch” on page 29. The procedure itself checks for interrupts and takes the highest priority interrupt that is pending.

The chkinterrupt() function for non-NMI levels returns one if:

- the current interrupt level is not masking the interrupt (CINTLEVEL < level)
- the interrupt is asserted (INTERRUPT)
- the corresponding interrupt enable is set (INTENABLE), and
- the interrupt is of the current level (LEVELMASK[level])

For NMI level interrupts, the no masking is done, but the edge sensor (made from NMIinput and lastNMIinput) is explicitly included to avoid repeating the NMI every cycle.

The takeinterrupt() function saves PC and PS in registers and changes them to take the interrupt.

```
procedure checkInterrupts()
     if chkinterrupt(NLEVEL+NNMI) then
          takeinterrupt[NLEVEL+NNMI]
     elseif chkinterrupt(NLEVEL+NNMI-1) then
     .
     .
     .
     elseif chkinterrupt(2) then
          takeinterrupt[2]
     elseif chkinterrupt(1) then
          Exception (Level1InterruptCause)
     endif
endprocedure checkInterrupts
```

where chkinterrupt and takeinterrupt are defined as:

```
function chkinterrupt(level)
```

```
        if level = NLEVEL+1 and NNMI = 1 then
             chkinterrupt ← NMIinput = 1 and LastNMIinput = 0
             lastNMIinput ← NMIinput
        elseif level ≤ NLEVEL then
             chkinterrupt ← (CINTLEVEL < level) and
                  ((LEVELMASK[level] and INTERRUPT and INTENABLE) ≠ 0)
        else
             chkinterrupt ← 0
        endif
   endfunction chkinterrupt

   function takeinterrupt(level)
        EPC[level] ← PC
        EPS[level] ← PS
        PC ← InterruptVector[level]
        PS.INTLEVEL ← level
        PS.EXCM ← 1
   endfunction takeinterrupt
```

4.4.6 Timer Interrupt Option

The Timer Interrupt Option is an in-core peripheral option for Xtensa processors. The Timer Interrupt Option can be used to generate periodic interrupts from a 32-bit counter and up to three 32-bit comparators. One counter period typically represents a number of seconds of elapsed time, depending on the clock rate at which the processor is configured.

- Prerequisites: Interrupt Option (page 100)
- Incompatible options: None

4.4.6.1 Timer Interrupt Option Architectural Additions

Table 4–77 and Table 4–78 show this option's architectural additions.

Table 4–77. Timer Interrupt Option Processor-Configuration Additions

Parameter	Description	Valid Values
NCCOMPARE	Number of 32-bit comparators	0..3[1,2]
TIMERINT[0..NCCOMPARE-1]	Interrupt number for each comparator	0..NINTERRUPT-1[3]

[1] The comparison registers can easily be multiplexed among multiple uses, so more than one comparator is usually not useful unless each comparator uses a different TIMERINT interrupt level.

[2] NCCOMPARE=0 with the Timer Interrupt Option specifies that CCOUNT exists, but there are no CCOMPARE registers or interrupts.

[3] NINTERRUPT is defined in the Interrupt Option, Table 4–70.

Table 4–78. Timer Interrupt Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
CCOUNT	1	32	Processor-clock count	R/W[2]	234
CCOMPARE	NCCOMPARE	32	Processor-clock compare (CCOUNT value at which an interrupt is generated)	R/W[3]	240-242

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 3–23 on page 46.

[2] This register is not normally written except after reset; it is writable primarily for testing purposes.

[3] Writing CCOMPARE clears a pending interrupt.

4.4.6.2 Clock Counting and Comparison

The CCOUNT register increments on every processor-clock cycle. When CCOUNT = CCOMPARE[i], a TIMERINT[i] interrupt request is generated. Although CCOUNT continues to increment and thus matches for only one cycle, the interrupt request is remembered until the interrupt is taken. In spite of this, timer interrupts are cleared by writing CCOMPARE[i], not by writing INTCLEAR. Interrupt configuration determines the interrupt number and level. It is automatically an Internal interrupt type (the INTTYPE[i] configuration parameter, Table 4–70).

For most applications, only one CCOMPARE register is required, because it can easily be shared for multiple uses. Applications that require a greater range of counting than that provided by the 32-bit CCOMPARE register can maintain a 64-bit cycle count and compare the upper bits in software.

CCOUNT and CCOMPARE[0..NCCOMPARE-1] are undefined after processor reset.

4.5 Options for Local Memory

The options in this section have the primary function of adding different kinds of memory, such as RAMs, ROMs, or caches to the processor. The added memories are tightly integrated into the processor pipeline for highest performance.

4.5.1 General Cache Option Features

This subsection describes general characteristics of caches that are referred to in multiple later subsections about specific cache options.

4.5.1.1 Cache Terminology

In the cache documentation a “line” is the smallest unit of data that can be moved between the cache and other parts of the system. If the cache is “direct-mapped,” each byte of memory may be placed in only one position in the cache. In a direct-mapped cache, the “index” refers to the portion of the address that is necessary to identify the cache line containing the access.

A cache is “set-associative” if there is more than one location in the cache into which any given line may be placed. It is “N-way set-associative” if there are N locations into which any given line may be placed. The set of all locations into which one line may be placed is called a “set” and the “index” refers to the portion of the address that is necessary to identify the set containing the access. The various locations within the set that are capable of containing a line are called the “ways” of the set. And the union of the Nth way of each set of the cache is the Nth “way” of the cache.

For example, a 4-way set-associative, 16k-byte cache with a 32-byte line size contains 512 lines. There are 128 sets of 4 lines each. The index is a 7-bit value that would most likely consist of Address<11:5> and is used to determine what set contains the line. The cache consists of 4 ways, each of which is 4k-bytes in size. A set represents 128 bytes of storage made up of four lines of 32 bytes each.

4.5.1.2 Cache Tag Format

Figure 4–14 shows the instruction- and data-cache tag format for Xtensa. The number of bits in the tag is a configuration parameter. So that all lines may be differentiated, the tag field must always be at least `32−log2(CacheBytes/CacheWayCount)` bits wide. If an MMU with pages smaller than a way of the cache is used, the tag field must also be at least `32−log2(MinPageSize)` bits wide. The actual tag field size is the maximum of these two values. The bits used in the tag field are the upper bits of the virtual address left justified in the register (the most significant bit of the register represents the most significant bit of the virtual address, bit 31). For example:

- A 16 kB direct-mapped cache would have an 18-bit tag field.
- A 16 kB 2-way associative cache would have a 19-bit tag field.
- A 16 kB 2-way associative cache in conjunction with an MMU with a 4kB minimum page size would have a 20-bit tag field.

The V bit is the line valid bit; 0 → invalid, 1 → valid. The three flag bits exist only for certain cache configurations. Any of the flag bits in Figure 4–14 not used in a particular configuration are reserved for future use and writing nonzero values to them gives undefined behavior. If the cache is set-associative, then bit[1] is the F bit and is used for cache miss refill way selection. If the cache is a data cache with writeback functionality, then the lowest remaining bit is the D bit, or dirty bit, and is used to signify whether the cache contains a value more recent than its backing store and must be written back. If the Index Lock Option is selected for that cache, the lowest remaining bit is the L bit, or lock bit, and is used to signify whether or not the line is locked and may not be replaced.[1]

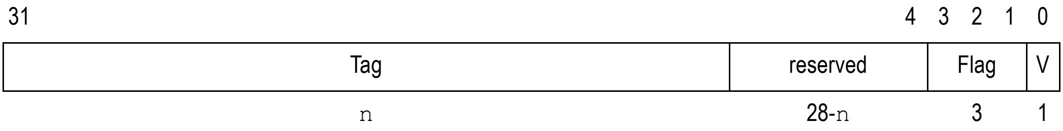


Figure 4–14. Instruction and Data Cache Tag Format for Xtensa

4.5.1.3 Cache Prefetch

There are two types of cache prefetch instructions. Normal prefetch instructions make no change in the architecturally visible state but simply attempt to move cache lines closer to the processor core. Any exception that might be raised causes the instruction to become a `NOP` rather than actually raising an exception. This allows prefetch instructions to be used without penalty in places where their addresses may not represent legal memory locations.

`IPF` attempts to move cache lines to the instruction cache. `DPFR`, `DPFRO`, `DPFW`, and `DPFWO` attempt to move cache lines to the data cache. The differences are that the `*R*` versions indicate that a write is not expected to the location in the immediate future while the `*W*` versions indicate that a write to the location is likely in the near future. The `*O` versions indicate that the most likely behavior is that the location is accessed in the near future, but that it is not worth keeping after that access as another access is not expected. `DPFWO` indicates that either a write or a read followed by a write is expected soon. The `*O` versions may be placed in different cache ways or kept in a separate buffer in some implementations.

The second type of prefetch instructions, prefetch and lock instructions, are only available under their respective Cache Index Lock Options. They also do not change the operation of memory loads and stores and they affect only cache tag state, which affects only future invalidation or line replacement operations on these lines. They are heavyweight operations and, unlike normal prefetch instructions, are only expected to be executed by code that sets up the caches for best performance.

The functions `iprefetch` and `dprefetch` are described below. Because they modify no architectural state, they are described only by comments.

[1] Note that the three flag bits are added sequentially from the right. The bits that exist are always contiguous with each other and with the V bit on the right. For the instruction cache, the valid combinations are 0-L-F, 0-0-F, and 0-0-0 because the instruction cache cannot be writeback and the Index Lock Option is only available for set-associative caches. For the data cache, the valid combinations are 0-L-F, 0-0-F, 0-0-0, L-D-F, 0-D-F, and 0-0-D, which are the same three with and without the dirty bit inserted in its order.

```
function iprefetch(vAddr, pAddr, lock)-- instruction prefetch
        if lock then
                -- move the line specified by vAddr/pAddr into the instruction cache
                -- mark the line locked
        else
                -- no architecturally visible operation performed
                -- no exception raised
                -- try to move the line specified by vAddr/pAddr into the instruction cache
        endif
endfunction iprefetch

function dprefetch(vAddr, pAddr, excl, once, lock)-- data prefetch
        if lock then
                -- move the line specified by vAddr/pAddr into the data cache
                -- mark the line locked
        else if excl then
                -- no architecturally visible operation performed
                -- no exception raised
                -- if caches are coherent, get an exclusive copy
                if once then
                        -- try to move the line specified by vAddr/pAddr where it can be
                        --      read and written once
                else
                        -- try to move the line specified by vAddr/pAddr into the data cache
                endif
        else
                -- no architecturally visible operation performed
                -- no exception raised
                if once then
                        -- try to move the line specified by vAddr/pAddr where it can be
read once
                else
                        -- try to move the line specified by vAddr/pAddr into the data cache
                endif
        endif
endfunction dprefetch
```

4.5.2 Instruction Cache Option

The Instruction Cache Option adds on-chip first-level instruction cache. The Instruction Cache Option also adds a few new instructions for prefetching and invalidation.

- Prerequisites: Processor Interface Option (page 194)
- Incompatible options: None

4.5.2.1 Instruction Cache Option Architectural Additions

Table 4–79 through Table 4–80 show this option's architectural additions.

Table 4–79. Instruction Cache Option Processor-Configuration Additions

Parameter	Description	Valid Values
`InstCacheWayCount`	Instruction-cache set associativity (ways)	1..4[1]
`InstCacheLineBytes`	Instruction-cache line size (bytes)	16, 32, 64, 128, 256[1]
`InstCacheBytes`	Instruction-cache size (bytes)	1kB, 1.5kB, 2kB, 3kB, ... 32kB[1]
`MemErrDetection`	Error detection type[2]	None, parity, ECC
`MemErrEnable`	Error enable	No-detect, detect[3]

[1] Valid values vary per implementation. Refer to information on local memories in a specific Xtensa processor data book.

[2] Must be identical for every instruction memory

[3] Detection may be enabled only when the Memory ECC/Parity Option is configured.

Table 4–80. Instruction Cache Option Instruction Additions

Instruction[1]	Format	Definition
`IPF`	RRI8	Instruction-cache prefetch This instruction checks whether the line containing the specified address is present in the instruction cache, and if not, begins the transfer of the line from memory to the cache. In some implementations, prefetching an instruction line may prevent the processor from taking an instruction cache miss later.
`IHI`	RRI8	Instruction-cache hit invalidate This instruction invalidates a line in the instruction cache if present and not locked. If the specified address is not in the instruction cache then this instruction has no effect. If the specified line is present and not locked, it is invalidated. This instruction is required before executing instructions that have been written by this processor, another processor, or DMA.
`III`	RRI8	Instruction-cache index invalidate This instruction uses the virtual address to choose a location in the instruction cache and invalidates the specified line if it is not locked. The method for mapping the virtual address to an instruction cache location is implementation-specific. This instruction is primarily useful for instruction cache initialization after power-up (note that if the Instruction Cache Index Lock Option is implemented, an `IIU` instruction should precede the `III`).

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243

See Section 5.7 “Caches and Local Memories” on page 240 for more information about synchronizations required when using the instruction cache.

4.5.3 Instruction Cache Test Option

The Instruction Cache Test Option is currently added to every processor that has an Instruction Cache Option; therefore, it is not actually a separate option. It adds instructions capable of reading and writing the tag and data of the instruction cache. These instructions are intended to be used in testing the instruction cache, rather than in operational code and may not be implemented in a binary compatible way in all future processors.

- Prerequisites: Processor Interface Option (page 194) and Instruction Cache Option (page 115)
- Incompatible options: None

4.5.3.1 Instruction Cache Test Option Architectural Additions

Table 4–81 shows this option’s architectural additions.

Table 4–81. Instruction Cache Test Option Instruction Additions

Instruction[1]	Format	Definition
LICT	RRR	Load instruction cache tag This instruction uses its address to specify a line in the Instruction Cache and loads the tag for that line into a register.
LICW	RRR	Load instruction cache word This instruction uses its address to specify a word in the instruction cache and loads that word into a register.
SICT	RRR	Store instruction cache tag This instruction uses its address to specify a line in the instruction cache and stores the tag for that line from a register.
SICW	RRR	Store instruction cache word This instruction uses its address to specify a word in the instruction cache and stores that word from a register.

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

The instruction-cache access instructions must be fetched from a region of memory that has the bypass attribute. Use an ISYNC instruction before transferring back to cached instruction space. See Section 5.7 “Caches and Local Memories” for more information about synchronizations required when using the instruction cache.

4.5.4 Instruction Cache Index Lock Option

The Instruction Cache Index Lock Option adds the capability of individually locking each line of the instruction cache. This option may only be added to a cache, which has two or more ways. One bit is added to the instruction cache tag RAM format. The Instruction Cache Index Lock Option also adds new instructions for locking and unlocking lines.

- Prerequisites: Processor Interface Option (page 194) and Instruction Cache Option (page 115)
- Incompatible options: None

4.5.4.1 Instruction Cache Index Lock Option Architectural Additions

Table 4–82 shows this option’s architectural additions.

Table 4–82. Instruction Cache Index Lock Option Instruction Additions

Instruction[1]	Format	Definition
`IPFL`	RRI4	Instruction-cache prefetch and lock This instruction checks whether the line containing the specified address is present in the instruction cache, and if not, begins the transfer of the line from memory to the cache. The line is placed in the instruction cache and the line marked as locked, that is, not replaceable by ordinary instruction cache misses. To unlock the line, use `IHU or IIU`. This instruction raises an illegal instruction exception on implementations that do not support instruction cache locking.
`IHU`	RRI4	Instruction-cache hit unlock This instruction unlocks a line in the instruction cache if present. If the specified address is not in the instruction cache then this instruction has no effect. If the specified line is present, it is unlocked. This instruction (or `IIU`) is required before invalidating a line if it is locked.
`IIU`	RRI4	Instruction-cache index unlock This instruction uses the virtual address to choose a location in the instruction cache and unlocks the specified line. The method for mapping the virtual address to an instruction cache location is implementation-specific. This instruction is primarily useful for unlocking the entire instruction cache. This instruction (or `IHU`) is required before invalidating a line if it is locked.

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

See Section 5.7 “Caches and Local Memories” for more information about synchronizations required when using the instruction cache.

4.5.5 Data Cache Option

The Data Cache Option adds on-chip first-level data cache. It supports prefetching, writing back, and invalidation.

The data-cache prefetch read/write/once instructions have been provided to improve performance, not to affect the processor state. Therefore, some implementations may choose to implement these instructions as no-op instructions. In general, the performance improvement from using these instructions is implementation-dependent. In some implementations, these instructions check whether the line containing the specified address is present in the data cache, and if not, begin the transfer of the line from memory.

- Prerequisites: Processor Interface Option (page 194)
- Incompatible options: None

4.5.5.1 Data Cache Option Architectural Additions

Table 4–83 and Table 4–84 show this option's architectural additions.

Table 4–83. Data Cache Option Processor-Configuration Additions

Parameter	Description	Valid Values
DataCacheWayCount	Data-cache set associativity (ways)	1..4[1]
DataCacheLineBytes	Data-cache line size (bytes)	16, 32, 64, 128, 256[1]
DataCacheBytes	Data-cache size (bytes)	1kB, 1.5kB, 2kB, 3kB, ... 32kB[1]
IsWriteback	Data-cache configured as writeback	Yes, No
MemErrDetection	Error detection type[2]	None, parity, ECC
MemErrEnable	Error enable	No-detect, detect[3]

[1] Valid values vary per implementation. Refer to information on local memories in a specific Xtensa processor data book.

[2] Must be identical for every data memory

[3] Detection may be enabled only when the Memory ECC/Parity Option is configured.

Table 4–84. Data Cache Option Instruction Additions

Instruction[1]	Format	Definition
DPFR, DPFW, DPFRO, DPFWO	RRI8	Data-cache prefetch {read,write}{,once} The four variants specify various "hints" about how the data is likely to be used in the future. DPFW and DPFWO indicate that the data is likely to be written in the near future. On some systems this is used to fetch the data with write permission (e.g. in a system with shared and exclusive states). DPFR and DPFRO indicate that the data is likely only to be read. The once forms, DPFRO and DPFWO, indicate that the data is likely to be read or written only once before it is replaced in the cache. On some implementations this might be used to select a specific cache way, or to select a streaming buffer instead of the cache.
DHWB	RRI8	Data-cache hit writeback If IsWriteback, this instruction forces dirty data in the data cache to be written back to memory. If the specified address is not in the data cache, or is present but unmodified, then this instruction has no effect. If the specified address is present and modified in the data cache, the line containing it is written back, and marked unmodified. This instruction is useful before a DMA read from memory, or to force writes to a frame buffer to become visible, or to force writes to memory shared by two processors. If not IsWriteback, DHWB is a no-op.

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243

Table 4–84. Data Cache Option Instruction Additions (continued)

Instruction[1]	Format	Definition
DHWBI	RRI8	Data-cache hit writeback invalidate If IsWriteback, this instruction forces dirty data in the data cache to be written back to memory. If the specified address is not in the data cache then this instruction has no effect. If the specified address is present and modified in the data cache, the line containing it is written back. After the writeback, if any, the line containing the specified address is invalidated if present and not locked. This instruction is useful in the same circumstances as DHWB and also before a DMA write to memory that does not completely overwrite the line. If not IsWriteback, DHWBI is identical to DHI except for privilege.
DIWB	RRI4	Data-cache Index writeback (added in T1050) If IsWriteback, this instruction forces dirty data in the data cache to be written back to memory. The virtual address is used, in an implementation dependent manner, to choose a cache line to write back. If the chosen line is unmodified, then this instruction has no effect. If the chosen line is modified in the data cache, the line containing it is written back, and marked unmodified. This instruction is useful for writing back the entire cache. If not IsWriteback, DIWB is a no-op.
DIWBI	RRI4	Data-cache index writeback invalidate (added in T1050) If IsWriteback, this instruction forces dirty data in the data cache to be written back to memory. The virtual address is used, in an implementation dependent manner, to choose a cache line to write back. If the chosen line is modified in the data cache, the line containing it is written back, and marked unmodified. After the writeback, if any, the chosen line is invalidated if it is not locked. This instruction is useful for writing back and invalidating the entire cache. If not IsWriteback, DIWBI simply invalidates without writeback.
DHI	RRI8	Data-cache hit invalidate This instruction invalidates a line in the data cache if present and not locked. If the specified address is not in the data cache then this instruction has no effect. If the specified address is present and not locked, it is invalidated. This instruction is useful before a DMA write to memory that overwrites the entire line.
DII	RRI4	Data-cache index invalidate This instruction uses the virtual address to choose a location in the data cache and invalidates the specified line if it is not locked. The method for mapping the virtual address to a data cache location is implementation-specific. This instruction is primarily useful for data cache initialization after power-up.

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243

See Section 5.7 “Caches and Local Memories” for more information about synchronizations required when using the data cache.

If IsWriteback, there is a dirty bit added to the data cache tag RAM format. The attributes described in Section 4.6.3.3 and Section 4.6.5.10 are then capable of setting a region of memory to be either write-back or write-through. If not IsWriteback, both attribute settings result in write-through semantics.

When a region of memory is marked write-back, any store that hits in the cache writes only the cache (setting the dirty bit, if it is not already set) and does not send a write on the PIF. Any store that does not hit in the cache causes a miss. When the line is filled, the semantics of a cache hit described above are followed. If a dirty line is evicted to use the space in the cache, the entire line will be written on the PIF. The DHWB, DHWBI, DIWB, and DIWBI instructions will also write back a line if it is marked dirty.

4.5.6 Data Cache Test Option

The Data Cache Test Option is currently added to every processor, which has a Data Cache Option and therefore, is not actually a separate option. It adds instructions capable of reading and writing the tag of the data cache. These instructions are intended to be used in testing the data cache, rather than in operational code and may not be implemented in a binary compatible way in all future processors.

- Prerequisites: Processor Interface Option (page 194) and Data Cache Option (page 118)
- Incompatible options: None

4.5.6.1 Data Cache Test Option Architectural Additions

Table 4–85 shows this option’s architectural additions.

Table 4–85. Data Cache Test Option Instruction Additions

Instruction[1]	Format	Definition
LDCT	RRR	Load data cache tag This instruction uses its address to specify a line in the instruction cache and loads the tag for that line into a register.
SDCT	RRR	Store data cache tag This instruction uses its address to specify a line in the instruction cache and stores the tag for that line from a register.

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

There are no instructions to access the data-cache data array. Normal loads and stores can be used for this purpose with the isolate attribute.

See Section 5.7 “Caches and Local Memories” for more information about synchronizations required when using the data cache.

4.5.7 Data Cache Index Lock Option

The Data Cache Index Lock Option adds the capability of individually locking each line of the data cache. One bit is added to the data cache tag RAM format. The Data Cache Index Lock Option also adds new instructions for locking and unlocking lines.

- Prerequisites: Processor Interface Option (page 194) and Data Cache Option (page 118)
- Incompatible options: None

4.5.7.1 Data Cache Index Lock Option Architectural Additions

Table 4–86 shows this option's architectural additions.

Table 4–86. Data Cache Index Lock Option Instruction Additions

Instruction[1]	Format	Definition
`DPFL`	RRI4	Data-cache prefetch and lock This instruction checks whether the line containing the specified address is present in the data cache, and if not, begins the transfer of the line from memory to the cache. The line is placed in the data cache and the line marked as locked, that is, not replaceable by ordinary data cache misses. To unlock the line, use `DHU` or `DIU`. This instruction raises an illegal instruction exception on implementations that do not support data cache locking.
`DHU`	RRI4	Data-cache hit unlock This instruction unlocks a line in the data cache if present. If the specified address is not in the data cache then this instruction has no effect. If the specified address is present, it is unlocked. This instruction (or `DIU`) is required before invalidating a line if it is locked.
`DIU`	RRI4	Data-cache index unlock This instruction uses the virtual address to choose a location in the data cache and unlocks the specified line. The method for mapping the virtual address to a data cache location is implementation-specific. This instruction is primarily useful for unlocking the entire data cache. This instruction (or `DHU`) is required before invalidating a line if it is locked.

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

See Section 5.7 "Caches and Local Memories" for more information about synchronizations required when using the data cache.

4.5.8 General RAM/ROM Option Features

The RAM and ROM options both provide internal memories that are part of the processor's address space and are accessed with the same timing as cache. These memories should not be confused with system RAM and ROM located outside of the processor, which are often larger, and may be used for both instructions and data, and shared between processors and other processing elements.

The basic configuration parameters are the size and base address of the memory. It is possible to configure cache, RAM, and ROM independently for both instruction and data, however some implementations may require an increased clock period if multiple instruction or multiple data memories are specified, or if the memory sizes are large. It is sometimes appropriate for the system designer to instead place RAMs and ROMs external to the processor and access these through the cache.

Every Instruction and Data RAM and ROM is always required to be naturally aligned (aligned on a boundary of a power of two which is equal to or larger than the size of the RAM/ROM) in physical address space. The mapping from virtual address space to physical address space must have the property that the Index bits of the RAM/ROM are identity mapped. This is a slightly less restrictive condition than requiring that the RAM/ROM must be contiguous and naturally aligned in virtual address space but this latter condition will always meet the requirement.

Instruction RAM can be referenced as data only by the `L32I`, `L32R` and `S32I` instructions and Instruction ROM referenced as data only by the `L32I and L32R` instructions. This functionality is provided for initialization and test purposes, for which performance is not critical, so these operations may be significantly slower on some Xtensa implementations. Most Xtensa code makes extensive use of `L32R` instructions, which load values from a location relative to the current `PC`. For this to perform well for code located in an instruction RAM or ROM, some sort of data memory (either internal or external) should be located within the 256 KB range of the `L32R` instruction or else the Extended L32R Option should be used.

Table 4–87 summarizes the restrictions on instruction and data RAM and ROM access. The exceptions listed assume no memory protection exception has already been raised on the access.

Table 4–87. RAM/ROM Access Restrictions

Memory	Instruction Fetch	L32R L32I L32I.N	Other Loads	S32I S32I.N	Other Stores
`InstROM`	ok	ok[1]	undefined	LSE[3]	LSE[3]
`InstRAM`	ok	ok[1]	undefined	ok[1]	undefined
`DataROM`	IFE[2]	ok	ok	LSE[3]	LSE[3]
`DataRAM`	IFE[2]	ok	ok	ok	ok
`UnifiedRAM`	ok	ok	ok	ok	ok

[1] Reduced performance on some Xtensa implementations

[2] Instruction fetch error exception

[3] Load store error exception

4.5.9 Instruction RAM Option

This option provides an internal, read-write instruction memory. It is typically useful as the only processor instruction store (no instruction cache) when all of the code for an application will fit in a small memory, or as an additional instruction store in parallel with the cache for code that must have constant access time for performance reasons.

- Prerequisites: None
- Incompatible options: None

4.5.9.1 Instruction RAM Option Architectural Additions

Table 4–88 shows this option’s configuration parameters. There are no processor state or instruction additions.

Table 4–88. Instruction RAM Option Processor-Configuration Additions

Parameter	Description	Valid Values
`InstRAMBytes`	Instruction RAM size (bytes)	512, 1kB, 2kB, 4kB, ... 256kB[1]
`InstRAMPAddr`	Instruction RAM base physical address	32-bit address, aligned on multiple of its size
`MemErrDetection`	Error detection type[2]	None, parity, ECC
`MemErrEnable`	Error enable	No-detect, detect[3]

[1] Refer to information on local memories in a specific Xtensa processor data book.

[2] Must be identical for every instruction memory

[3] Detection may be enabled only when the Memory ECC/Parity Option is configured.

Instruction RAM may be accessed as data using the L32I, L32R, and S32I instructions. The operation of other loads and stores on InstRAM addresses is not defined. S32I is useful for copying code into the InstRAM; L32I is useful for diagnostic testing of InstRAM, and L32R allows constants to be loaded from InstRAM if no data memory is within range. While L32I, L32R, and S32I to InstRAM are defined, on many implementations these accesses are much slower than references to data RAM, ROM, or cache, and thus the use of InstRAM for data storage is not recommended.

4.5.10 Instruction ROM Option

This option provides an internal, read-only instruction memory. It is typically useful as the only processor instruction store (no instruction cache) when all of the code for an application will fit in a small memory, or as an additional instruction store in parallel with the cache for code that must have constant access time for performance reasons. Because ROM is read-only, only code that is not subject to change should be put here.

- Prerequisites: None
- Incompatible options: None

4.5.10.1 Instruction ROM Option Architectural Additions

Table 4–89 shows this option's configuration parameters. There are no processor state or instruction additions.

Table 4–89. Instruction ROM Option Processor-Configuration Additions

Parameter	Description	Valid Values
InstROMBytes	Instruction ROM size (bytes)	512, 1kB, 2kB, 4kB, ... 256kB[1]
InstROMPAddr	Instruction ROM base physical address	32-bit address, aligned on multiple of its size

[1] Refer to information on Local Memories in a specific Xtensa processor data book.

Instruction ROM may be accessed as data using the L32I and L32R instructions. The operation of other loads on InstROM addresses is not defined. L32I is useful for diagnostic testing of InstROM, and L32R allows constants to be loaded from InstROM if no data memory is within range. While L32I and L32R to InstROM are defined, on many implementations these accesses are much slower than references to data RAM, ROM, or cache, and thus the use of InstROM for data storage is not recommended.

4.5.11 Data RAM Option

This option provides an internal, read-write data memory. It is typically useful as the only processor data store (no data cache) when all of the data for an application will fit in a small memory, or as an additional data store in parallel with the cache for data that must be constant access time for performance reasons.

- Prerequisites: None
- Incompatible options: None

4.5.11.1 Data RAM Option Architectural Additions

Table 4–90 shows this option's configuration parameters. There are no processor state or instruction additions.

Table 4–90. Data RAM Option Processor-Configuration Additions

Parameter	Description	Valid Values
`DataRAMBytes`	Data RAM size (bytes)	512, 1kB, 2kB, 4kB, ... 256kB[1]
`DataRAMPAddr`	Data RAM base physical address	32-bit address, aligned on multiple of its size
`MemErrDetection`	Error detection type[2]	None, parity, ECC
`MemErrEnable`	Error enable	No-detect, detect[3]

[1] Refer to information on Local Memories in a specific Xtensa processor data book.

[2] Must be identical for every data memory

[3] Detection may be enabled only when the Memory ECC/Parity Option is configured.

In the absence of the Extended L32R Option it is recommended that processors with data RAM or ROM and no data cache be configured with the DataRAMPAddr or DataROMPAddr below the lowest instruction address and above the highest instruction address minus 256 KB, so that the `L32R` literals can be stored in RAM or ROM for fast access. The processor will fetch `L32R` literals from the instruction RAM, or ROM, but in many implementations several cycles are required for the fetch, making the use of this feature undesirable. The Extended L32R Option allows less restricted placement.

4.5.12 Data ROM Option

This option provides an internal, read-only data memory. It is typically useful as an additional data store in parallel with the cache for data that must be constant access time for performance reasons.

- Prerequisites: None
- Incompatible options: None

4.5.12.1 Data ROM Option Architectural Additions

Table 4–91 shows this option’s configuration parameters. There are no processor state or instruction additions.

Table 4–91. Data ROM Option Processor-Configuration Additions

Parameter	Description	Valid Values
`DataROMBytes`	Data ROM size (bytes)	512, 1kB, 2kB, 4kB, ... 256kB[1]
`DataROMPAddr`	Data ROM base physical address	32-bit address, aligned on multiple of its size

[1] Refer to information on local memories in a specific Xtensa processor data book.

4.5.13 XLMI Option

The XLMI Option, or Xtensa Local Memory Interface Option, allows the attachment of hardware other than caches, RAMs, and ROMs into the pipeline of the processor rather than on the processor interface bus. The advantage of the XLMI is that the latency is lower. The disadvantage is that speculation must be explicitly allowed for on loads. The XLMI port contains signals that inform external devices after the fact concerning whether a load was or was not speculative. Stores are never speculative. Refer to a specific Xtensa processor data book for more detail.

- Prerequisites: None
- Incompatible options: None

Instructions may not be fetched from an XLMI interface. The virtual and physical addresses of the entire XLMI region must be identical in all bits.

4.5.13.1 XLMI Option Architectural Additions

Table 4–92 shows this option’s configuration parameters. There are no processor state or instruction additions.

Table 4–92. XLMI Option Processor-Configuration Additions

Parameter	Description	Valid Values
`XLMIBytes`	XLMI size (bytes)	512, 1kB, 2kB, 4kB, ... 256kB[1]
`XLMIPAddr`	XLMI base physical address	32-bit address, aligned on multiple of its size

[1] Refer to information on local memories in a specific Xtensa processor data book.

4.5.14 Hardware Alignment Option

The Hardware Alignment Option adds hardware to the processor which allows loads and stores to work correctly at any arbitrary alignment. It does this by making multiple accesses where necessary and combining the results. Unaligned accesses are still slower than aligned accesses, but this option is more efficient than the Unaligned Exception Option with software handler. In addition, the Hardware Alignment Option will work in situations where a software handler is difficult to write (for example, a load and operate instruction).

- Prerequisites: Unaligned Exception Option (page 99)
- Incompatible options: None

The Hardware Alignment Option builds on the Unaligned Exception Option so that almost all potential `LoadStoreAlignmentCause` exceptions are handled transparently by hardware instead. A few situations, which are never expected to happen in real software, still raise a `LoadStoreAlignmentCause` exception. In order to properly handle all TLB misses and other exceptions, the priority of the `LoadStoreAlignmentCause` exception is lower when the Hardware Alignment Option is present than when it is not. Exception priorities are listed in Section 4.4.1.11.

A `LoadStoreAlignmentCause` exception may still be raised in some implementations with the Hardware Alignment Option if the address of a load or store instruction is not a multiple of its size and any of the following conditions is also true:

- The instruction is one of `L32AI`, `S32RI`, or `S32C1I`.
- The memory type for either portion is XLMI, IRAM, or IROM.
- The memory types (cache, DataRAM, bypass) of the two portions differ.
- The cache attribute for either portion is Isolate.
- The column labeled "Meaning for Cache Access" in either Table 4–104 on page 155 or Table 4–109 on page 178 is different for the two portions of the access.

4.5.15 Memory ECC/Parity Option

The Memory ECC/Parity Option allows the local memories and caches of Xtensa processors to be protected against errors by either parity or error correcting code (ECC). It does not affect the processor interface and system memories must maintain their own error detection and correction. Local memories must be wide enough to contain the additional bits required. The generation and checking of parity or ECC is done in the Xtensa core through a combination of hardware and software mechanisms.

- Prerequisites: Exception Option (page 82)
- Incompatible options: None

Each memory may be protected or not protected individually. All protected instruction memories must use a single protection type (parity or ECC). Likewise, all protected data memories must use a single protection type. For parity protection, data memories require one additional bit per byte while instruction memories require one additional bit per four bytes and cache tags require one additional bit per tag. For ECC protection, instruction memories require 7 additional bits per 32-bit word, data memories require 5 additional bits per byte, and cache tags require 7 additional bits per tag.

The core computes parity or ECC bits on every store without doing a read-modify-write. On every load or instruction fetch, these bits are checked and an exception is raised for parity errors or for uncorrectable ECC errors. For correctable errors, a control bit in the memory error status register (Table 4–94) indicates whether to raise an exception or simply correct the value to be used (but not the value in memory) and continue. In addition, correctable ECC errors assert an output pin which may be used as an interrupt. Implementations may or may not implement hardware correction. If they do not implement it, the exception is always raised.

4.5.15.1 Memory ECC/Parity Option Architectural Additions

Table 4–93 through Table 4–95 show this option's architectural additions.

Table 4–93. Memory ECC/Parity Option Processor-Configuration Additions

Parameter	Description	Valid Values
`MemoryErrorVector`	Exception vector for memory errors	32-bit address
	Each RAM/Cache has configuration additions valid when the Memory ECC/Parity Option is configured	

Table 4–94. Memory ECC/Parity Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Access
`MEPC`	1	32	Memory error PC register	R/W	106
`MEPS`	1	same as `PS` register[1]	Memory error PS register	R/W	107
`MESAVE`	1	32	Memory error save register	R/W	108
`MESR`	1	19	Memory error status register	R/W	109
`MECR`	1	22	Memory error check register	R/W	110
`MEVADDR`	1	32	Memory error virtual address register	R/W	111

[1] There are enough bits to save all configured PS Register Fields. See Table 4–63 on page 87.

Table 4–95. Memory ECC/Parity Option Instruction Additions

Instruction[1]	Format	Definition
`RFME`	RRR	Return from memory error

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.5.15.2 Memory Error Information Registers

Three registers are used to maintain information about a memory error. They are updated for memory errors which do not raise an exception, as well as those which do. The memory error status register (`MESR`), shown in Figure 4–15 with further description in Table 4–98, contains control bits that control the operation of memory errors and status bits that hold information about memory errors that have occurred.

Under normal operation, check bits are always calculated and written to local memories. When ECC is enabled, an uncorrectable error, or a correctable error for which the `MESR.DataExc` or `MESR.InstExc` bit is set, will raise an exception whenever it is encountered during either a load or a dirty castout. Inbound PIF operations return an error when appropriate but the error will not be noted by the local processor. Correctable errors during a dirty castout when `MESR.DataExc` is clear may, in some implementations, correct the error on the fly without setting `MESR.RCE` or associated status.

When ECC is enabled and either the `MESR.DataExc` bit or the `MESR.InstExc` bit is clear or the `MESR.MemE` bit is set, hardware may be able to correct an error without raising an exception. This may cause `MESR.RCE` (along with many other fields), `MESR.DLCE`, or `MESR.ILCE` to be set by hardware at an arbitrary time.

In addition, an external pin reflects the state of `MESR.RCE` and can be connected to an interrupt input on the Xtensa processor itself or on another processor. This interrupt may be at a much lower priority than the memory error exception handler, but it can still repair the memory itself and/or log the error much as the memory error exception handler might. `MESR.RCE` must be cleared by software to return the external pin to zero and to re-arm the mechanism for recording correctable errors.

31 30	29 28	27 24	23 22	21 20	19 18	17 16	15 12	11	10	9	8	7	6	5	4	3 2	1	0
Error Type	*	Memory Type	*	Acc. Type	*	Way Numb.	*	InstExc	DataExc	ErrTest	ErrEnab	*	ILCE	DLCE	RCE	*	DME	MemE
2	2	4	2	2	2	2	4	1	1	1	1	1	1	1	1	2	1	1

Figure 4–15. MESR Register Format

Table 4–96. MESR Register Fields

Field	Width (bits)	Definition
MemE	1	Memory error. 0 → Memory error exception not in progress. 1 → Memory error exception in progress. Set on taking memory error exception. Cleared by RFME instruction. Software reads and writes MemE normally.
DME	1	Double Memory error. 0 → Normal memory error exception. 1 → Current memory error exception encountered during a Memory error exception. Set on taking memory error exception while MemE is set. Hardware does not clear. Software reads and writes DME normally.
RCE (ECC[1])	1	Recorded correctable error. (Exists only if ECC is configured.) 0 → Status refers to something else. 1 → Status refers to an error corrected by hardware. RCE means that status refers to a correctable memory error that has been fixed in hardware. Status, here, means the group of state that contains information about a memory error. It consists of the status fields of MESR (Way Number, Access Type, Memory Type, and Error Type) and the contents of the MECR and MEVADDR registers. The recorded information may be used to fix the error in the memory copy or to log the error. RCE is set by hardware whenever MemE is clear, RCE is clear, and a correctable error is fixed in hardware. RCE is cleared by hardware when a memory exception is raised as the recorded information is lost and either DLCE or ILCE is set in its place. Software reads and writes RCE normally.
DLCE (ECC[1])	1	Data lost correctable error. (Exists only if ECC is configured.) 0 → No information has been lost about data hardware corrected memory errors. 1 → Information has been lost about data hardware corrected memory errors. DLCE means that there has been a correctable error on a data (execute) access which has not been recorded because 1) it happened during a memory error exception (MemE set), 2) a memory error exception happened before it was recorded (RCE now cleared), or 3) it happened after another correctable error and before that error was recorded (RCE also set). DLCE is set by hardware whenever any data (execute) correctable error is fixed in hardware but MemE or RCE is set and the new Access Type is not instruction fetch. DLCE is also set by hardware when any memory exception is raised with RCE set and with the current Access Type is not instruction fetch. DLCE is never cleared by hardware. Software reads and writes DLCE normally.

[1] In some implementations the bits used with ECC may exist as state bits without effect even when only parity is configured.

Table 4–96. MESR Register Fields (continued)

Field	Width (bits)	Definition
ILCE (ECC[1])	1	Instruction fetch (Ifetch) lost correctable error. (Exists only if ECC is configured.) 0 → No information has been lost about ifetch hardware corrected memory errors. 1 → Information has been lost about ifetch hardware corrected memory errors. ILCE means that there has been a correctable error on an Ifetch access which has not been recorded because 1) it happened during a memory error exception (MemE set), 2) a memory error exception happened before it was recorded (RCE now cleared), or 3) it happened after another correctable error and before that error was recorded (RCE also set). ILCE is set by hardware whenever any Ifetch correctable error is fixed in hardware but MemE or RCE is set and the new Access Type is instruction fetch. ILCE is also set by hardware when any memory exception is raised with RCE set and with the current Access Type is instruction fetch. ILCE is never cleared by hardware. Software reads and writes ILCE normally.
ErrEnab	1	Enable Memory ECC/Parity Option errors. 0 → Memory errors are disabled. 1 → Memory errors are enabled. When ErrEnab is set, memory error exceptions and corrections are enabled. When ErrEnab is clear, the same values are written to memories, but no checks and no exceptions are raised on memory reads. Operation is undefined when both ErrEnab and ErrTest are set. ErrEnab is not modified by hardware.
ErrTest	1	Memory error test mode. 0 → Normal memory error operation. 1 → Special memory error test operation. When ErrTest is set, the memory write instructions S32I, S32I.N, SICT, SICW, and SDCT insert the actual contents of the MECR register into the memory check bits and the memory read instructions L32I, L32I.N, LICT, LICW, and LDCT always place the actual check bits read from memory into the MECR register. The operation of other memory access instructions is undefined when ErrTest is set. When ErrTest is clear, memory writes compute appropriate check bits for each write and memory reads do not affect the MECR register (unless a memory error is detected). Cache fills and Inbound PIF operations are unaffected by the setting of the ErrTest bit. Operation is undefined when both ErrEnab and ErrTest are set. ErrTest is not modified by hardware.

[1] In some implementations the bits used with ECC may exist as state bits without effect even when only parity is configured.

Table 4–96. MESR Register Fields (continued)

Field	Width (bits)	Definition
DataExc (ECC[1])	1	Data exception. (Exists only if ECC is configured.) 0 → No exception on hardware correctable data memory errors. 1 → Memory error exception on hardware correctable data memory errors. Set by software to cause memory errors which might be handled in hardware on data accesses to raise the memory error exception instead. This bit is forced to 1 (cannot be cleared) if hardware is unable to handle any data access errors. If MemE is set, no exception is raised for errors which hardware can handle even if DataExc is set. DataExc is not modified by hardware.
InstExc (ECC[1])	1	Instruction exception. (Exists only if ECC is configured.) 0 → No exception on hardware correctable instruction fetch memory errors. 1 → Memory error exception on hardware correctable instr. fetch memory errors. Set by software to cause memory errors which might be handled in hardware on instruction fetches to raise the memory error exception instead. This bit is forced to 1 (cannot be cleared) if hardware is unable to handle any instruction fetch errors. If MemE is set, no exception is raised for errors which hardware can handle even if InstExc is set. InstExc is not modified by hardware.
Way Number	2	Cache way number of a memory error. (Exists only if a multiway cache is configured.) When RCE or MemE is set and the Memory Type field points to a cache, this field contains the cache way number containing the error. Way Number is set by hardware whenever MemE is clear, RCE is clear, and a correctable error is fixed in hardware or whenever a memory exception is raised.
Access Type	2	Access type of an access with memory error. 0 → Memory error during load or store 1 → Memory error during instruction fetch 2 → Memory error during instruction memory access (such as IPFL or IHI) 3 → Memory error during dirty line castout When RCE or MemE is set, this field contains an indication of the access type which caused the memory error. Access Type is set by hardware whenever MemE is clear, RCE is clear, and a correctable error is fixed in hardware or whenever a memory exception is raised.

[1] In some implementations the bits used with ECC may exist as state bits without effect even when only parity is configured.

Table 4–96. MESR Register Fields (continued)

Field	Width (bits)	Definition
Memory Type	4	Memory type to which the access with memory error was directed. 0 → Error in instruction RAM 0. 1 → Error in data RAM 0. 2 → Error in instruction cache data array. 3 → Error in data cache data array 4 → Error in instruction RAM 1. 5 → Error in data RAM 1. 6 → Error in Instruction cache tag array. 7 → Error in data cache tag array 8-15 → Reserved When RCE or MemE is set, this field contains a pointer to the memory which caused the memory error. Memory Type is set by hardware whenever MemE is clear, RCE is clear, and a correctable error is fixed in hardware or whenever a memory exception is raised.
Error Type	2	Type of memory error. 0 → Reserved 1 → Parity error 2 → Correctable ECC error 3 → Uncorrectable ECC error When RCE or MemE is set, this field contains an indicator of the type of memory error which caused the memory error. Error Type is set by hardware whenever MemE is clear, RCE is clear, and a correctable error is fixed in hardware or whenever a memory exception is raised.
*		Reserved for future use Writing a non-zero value to one of these fields results in undefined processor behavior. These bits read as undefined.

[1] In some implementations the bits used with ECC may exist as state bits without effect even when only parity is configured.

The memory error check register (MECR), shown in Figure 4–16 with further description in Table 4–97, contains syndrome bits that indicate what error occurred. For data memories, all four check fields are used so that all bytes may be covered. For instruction memories or for cache tags, only the Check 0 field is used.

When the ErrEnab bit of the MESR register is set and the RCE or MemE bit of the MESR register is turned on, this register contains error syndromes. For parity memories, the error syndrome is ’1’ corresponding to a parity error and ’0’ corresponding to no parity error. For ECC memories, the error syndrome is a set of bits equal in length to the number of check bits associated with that portion of memory. The bits are all zero where there is no error. Non-zero values give more information about which bit or bits are in error. The exact encoding depends on the implementation. See the *Xtensa Microprocessor Data Book* for more information on the encoding.

When the `ErrTest` bit of the `MESR` register is set, `MECR` is loaded by every `L32I`, `L32I.N`, `LICT`, `LICW`, and `LDCT` instruction with the actual check bits which have been read from memory. When the `ErrTest` bit of the `MESR` register is set, the fields of `MECR` are used by the `S32I`, `S32I.N`, `SICT`, `SICW`, and `SDCT` instructions to write the memory check bits. Operation of other memory access instructions is not defined when `ErrTest` is set. Operation is not defined if both `ErrEnab` and `ErrTest` are set.

Error addresses are reported with reference to the 32-bit word containing the error regardless of the size of the access and for all errors `MEVADDR` contains an address aligned to 32-bits. For data memories, the check field(s) in `MECR` corresponding to the damaged byte(s) contains a non-zero syndrome. For tag memories and instruction memories, the Check 0 field of `MECR` contains the syndrome for the entire word. Errors in portions of the word not actually used by the access may or may not be reported in `MECR`.

31 29	28 24	23 21	20 16	15 13	12 8	7	6 0
*	Check 3	*	Check 2	*	Check 1	*	Check 0
3	5	3	5	3	5	1	7

Figure 4–16. MECR Register Format

Table 4–97. MECR Register Fields

Field	Width (bits)	Definition
`Check 3`	5	Check bits for the high order byte of a 32 bit data word. This field is valid for accesses to data RAM and data cache. It contains 5 check bits for ECC memories and 1 check bit (at the right end of the field) for parity memories. The field is associated with the highest address byte in little endian processors and the lowest address byte in big endian processors.
`Check 2`	5	Check bits for the next high order byte of a 32 bit data word. This field is valid for accesses to data RAM and data cache. It contains 5 check bits for ECC memories and 1 check bit (at the right end of the field) for parity memories. The field is associated with the second highest address byte in little endian processors and the second lowest address byte in big endian processors.

Table 4–97. MECR Register Fields (continued)

Field	Width (bits)	Definition
`Check 1`	5	Check bits for the next low order byte of a 32 bit data word. This field is valid for accesses to data RAM and data cache. It contains 5 check bits for ECC memories and 1 check bit (at the right end of the field) for parity memories. The field is associated with the second lowest address byte in little endian processors and the second highest address byte in big endian processors.
`Check 0`	7	Check bits for the low order byte of a 32 bit data word. For accesses to data RAM and data cache this field contains 5 check bits for ECC memories and 1 check bit (at the right end of the field) for parity memories and is associated with the lowest address byte in little endian processors and the highest address byte in big endian processors. For accesses to instruction RAM, instruction cache and all cache tags, this field contains 7 check bits for ECC memories and 1 check bit (at the right end of the field) for parity memories and covers the whole 32-bit word or tag.
*		Reserved for future use Writing a non-zero value to one of these fields results in undefined processor behavior. These bits read as undefined.

The memory error virtual address register (`MEVADDR`), shown in Figure 4–17, contains address information regarding the location of the error. Table 4–98 details its contents as a function of two fields of the `MESR` register. For errors in cache tags and for errors in castout data, `MEVADDR` contains only index information. Along with the `Way Number` field in `MESR`, this allows the incorrect memory bits to be located. For errors in instructions or data being accessed, `MEVADDR` contains the full virtual address used by the instruction. Along with other status information, `MEVADDR` is written when the `ErrEnab` bit of the `MESR` register is set and the `RCE` or `MemE` bit of the `MESR` register is turned on.

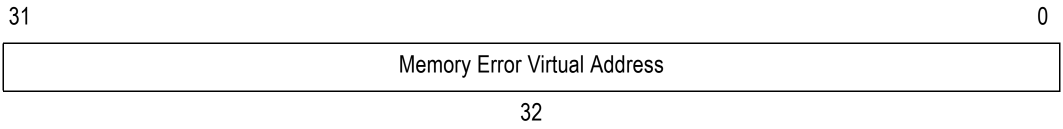


Figure 4–17. MEVADDR Register Format

Table 4–98. MEVADDR Contents

MESR Memory Type	MESR Access Type	MEVADDR Contents
Instruction RAM *n*		Full virtual address used in instruction.
Data RAM *n*		Full virtual address used in instruction.

1. For LICW instructions or Isolate cache attributes, only the index and way bits along with lower order bits are valid.

Table 4–98. MEVADDR Contents (continued)

MESR Memory Type	MESR Access Type	MEVADDR Contents
Instruction cache tag array		Index bits are valid, other bits are undefined.
Instruction cache data array		Full virtual address used in instruction.[1]
Data cache tag array		Index bits are valid, other bits are undefined.
Data cache data array	LoadStore	Full virtual address used in instruction.[1]
Data cache data array	Castout	Index bits are valid, other bits are undefined.

[1] For LICW instructions or Isolate cache attributes, only the index and way bits along with lower order bits are valid.

4.5.15.3 The Exception Registers

Three of the new registers created by this option are used in order to be able to take a memory error exception at any time and return. As an exception, memory error cannot be masked except by the `MESR.ErrEnab` bit. Whenever the exception is taken, the PC of the instruction taking the error is saved in the `MEPC` register, the PS register is saved in the `MEPS` register, and the `MESAVE` register is available for software use in the exception handler.

When an actual memory error exception is taken, the `MEPC` and `MEPS` registers are loaded with the original values of `PC` and `PS`, and then `PS.INTLEVEL` is raised to `NLEVEL` so that all interrupts except NMI are masked and the `PS.EXCM` bit is set so that an ordinary exception will cause a double exception. When hardware corrects a correctable memory error, these actions are not taken, allowing memory error corrections even in the memory error exception handler.

A memory error exception may be taken at any time. This means that, even without hardware correction, a memory error can be handled any time except during a memory error handler. With hardware correction, only an uncorrectable memory error taken during a handler for another uncorrectable memory error is fatal.

4.5.15.4 Memory Error Semantics

Memory errors have the following semantics:

```
procedure MemoryError
     return if !MESR.ErrEnab
     exc ← ParityError | UncorrectableECCError
     exc ← 1 if !MESR.MemE & MESR.InsExc & AccessType = IFetch
     exc ← 1 if !MESR.MemE & MESR.DatExc & AccessType ≠ IFetch
     MESR.ILCE ← 1 if exc & MESR.RCE & MESR.AccessType = IFetch
     MESR.DLCE ← 1 if exc & MESR.RCE & MESR.AccessType ≠ IFetch
     MESR.ILCE ← 1 if !exc & MESR.RCE & AccessType = IFetch
     MESR.DLCE ← 1 if !exc & MESR.RCE & AccessType ≠ IFetch
```

```
        MESR.ILCE ← 1 if !exc & MESR.MemE & AccessType = IFetch
        MESR.DLCE ← 1 if !exc & MESR.MemE & AccessType ≠ IFetch
        if exc | !MESR.RCE then
             MESR.WayNumber ← WayNumber
             MESR.AccessType ← AccessType
             MESR.MemoryType ← MemoryType
             MESR.ErrorType ← ErrorType
             MECR ← CheckBits
             if MESR.AccessType = Castout then
                  MEVADDR ← Undefined||CacheIndex||Undefined
             elsif MESR.MemoryType = Tag then
                  MEVADDR ← Undefined||CacheIndex||Undefined
             else
                  MEVADDR ← VAddr
             endif
             MESR.RCE ← !exc
        endif
        if exc then
             MESR.DME ← MESR.MemE
             MESR.MemE ← 1
             MEPC ← PC
             MEPS ← PS
             nextPC ← MemoryErrorExceptionVector
             PS.INTLEVEL ← NLEVEL
             PS.EXCM ← 1
        endif
   endprocedure MemoryError
```

4.6 Options for Memory Protection and Translation

Xtensa processors employ one of the options in this section for memory protection and translation. The introduction in Section 4.6.1 provides background information for the options in this section. The Region Protection Option described in Section 4.6.3 provides control of memory by 512 MB regions. Within each region, accessibility, cacheability, and characteristics of cacheability can be controlled. The Region Translation Option described in Section 4.6.4 builds on that and adds a translation table with an entry for each region so that virtual addresses in that region can be translated to corresponding physical addresses in any of the 512 MB regions. The MMU Option described in Section 4.6.5 is a full paging memory management unit. It supports hardware refill of the TLB from page tables in memory.

4.6.1 Overview of Memory Management Concepts

Section 4.6.1.1 gives an overview of the basic memory translation scheme used in Xtensa processors. Section 4.6.1.2 gives an overview of the basic memory protection scheme used in Xtensa processors, and Section 4.6.1.3 gives an overview of the concept of attributes. These subsections take a broader view of the overall process and indicate the direction future memory protection and translation options may take.

4.6.1.1 Overview of Memory Translation

This subsection presents an overview of the thinking behind the memory translation in the available options. It also provides insight into the kinds of extensions that are likely in the future.

The available memory protection and translations options that support virtual-to-physical address translation do so via an instruction TLB and a data TLB. (“TLB” was originally an acronym for translation lookaside buffer, but this meaning is no longer entirely accurate; in this document TLB simply means the translation hardware.) These two hardware structures may, in some configurations, act as translation caches that are refilled by hardware from a common page table structure in memory. In other configurations, a TLB may be self-sufficient for its translations, and no page tables are required.

A TLB consists of several entries, each of which maps one page (the page size may vary with each entry). Virtual-to-physical address translation consists of searching the TLB for an entry that matches the most significant bits of the virtual address and replacing those bits with bits from the TLB entry. The least significant bits of the virtual address are identical between the virtual and physical addresses. The translation input and output are called the virtual page number (VPN) and the physical page number (PPN) respectively. The TLB search also involves matching the address space identifier (ASID) bits of the TLB entry to one of the current ASIDs stored in the `RASID` register (more on this below). The number of bits not translated is determined by the page size, which can be dynamically programmed from a set of configuration specified values. The TLB entry also supplies some attribute bits for the page, including bits that determine the cacheability of the page’s data, whether it is writable or not, and so forth. This is illustrated in Figure 4–18.

It is illegal for more than one TLB entry to match both the virtual address and the ASID. This is true even if the entries have different ASIDs which match at different ring levels. Software is responsible for making sure the address range of all TLB entries visible according to the ASID values in the `RASID` register never overlap. Implementations may detect this situation and take a MultiHit exception in this situation to aid in debugging.

The instruction and data TLBs can be configured independently for most parameters, which is appropriate because the instruction and data references of processors can have fairly different requirements, and in some systems additional flexibility may be appropriate on one but not the other. However, when the two TLBs both refill from the common memory page table, the associated parameters are shared.

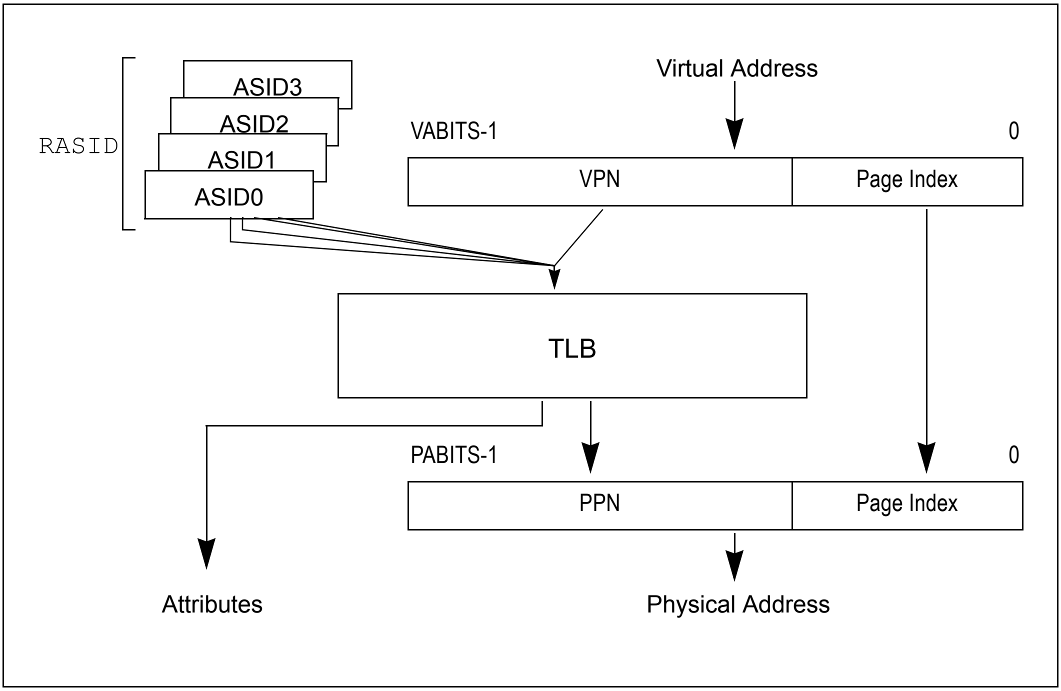


Figure 4–18. Virtual-to-Physical Address Translation

Xtensa implementations may perform virtual-to-physical address translation in parallel or series with cache, RAM, ROM, and XLMI access. However, the translated physical address is always used to decide which cache, RAM, or ROM access to use. Thus caches are potentially virtually indexed, even though they are always physically tagged. When the number of cache index bits (that is `log2(CacheBytes/WayCount)`) is greater than a page index and the same physical memory is mapped at multiple virtual addresses, there is the possibility of multiple cache locations being used for the same physical memory line, which can lead to the multiple views of memory being inconsistent. In such a system, software typically avoids this situation by restricting the virtual addresses for multiply mapped physical memory. This software restriction is often referred to as “page coloring.” If physically indexed caches are necessary (and generally they are not), the system designer may configure the TLBs such that cache index is a physical address by using a large page size or a high cache associativity so that the cache index bits are within the portion of the virtual and physical addresses that are identical.

The TLBs are N-way set-associative structures with heterogeneous “ways” and a configurable N. Each way has its own parameters, such as the number of entries, page size(s), constant or variable virtual address, and constant or variable physical address and attributes. It is the ability to specify constant translations in some or all of the ways that allows Xtensa’s TLBs to span smoothly from a fixed memory map to a fully programmable one. Fully or partially constant entries can be converted to logic gates in the TLB at significantly lower cost than a run-time programmable way. In addition, even processors with generally programmable MMUs often have a few hardwired translations. Xtensa can easily represent these hardwired translations with its constant TLB entries. Xtensa actually requires a few constant TLB entries to provide translation in some circumstances, such as at reset and during exception handling.

The virtual address input to the TLBs is actually the catenation of an address space identifier (ASID) specified in a processor register with the 32-bit virtual address from the fetch, load, or store address calculation. ASIDs allow software to change the address space seen by the processor (for example, on a context switch) with a simple register write without changing the TLB contents. The TLB stores an ASID with each entry, and so can simultaneously hold translations for multiple address spaces. The number of ASID bits is configurable. ASIDs are also an integral part of protection, as they specify the accessibility of memory by the processor at different privilege levels, as described in the next section.

Xtensa TLBs do not have a separate valid bit in each entry. Instead, a reserved ASID value of 0 is used to indicate an invalid entry. This can be viewed as saving a bit, or as almost doubling the number of ASIDs for the same number of hardware bits stored in a TLB entry.

Non-constant ways may be configured as AutoRefill. If no entry matching an access is found in a TLB with one or more AutoRefill ways, the processor will attempt to load a page table entry (PTE) from memory and write it into an entry of one of the AutoRefill ways. A TLB with no AutoRefill ways does not use the page table.

Each way of a TLB is configured with a list of page sizes (expressed as the number of bits in a page index). If the list has one element, the page size for that way is fixed. If the list has more than one element, the page size of the way may be varied at runtime via the `ITLBCFG` or `DTLBCFG` registers. When AutoRefill ways have programmable page size, the PTE has a page size field (the value is an index into the `PTEPageSizes` configuration parameter), and hardware refill restricts the refill way selection to ways programmed with a page size matching the page size in the PTE. When looking up an address in the TLB, each way’s page size determines which bits are used to select one of the way’s entries for comparison: $\texttt{vAddr}_{P+\log2(\text{IndexCount})-1..P}$ is the way index where P is the number of bits configured or programmed for the way page size.

4.6.1.2 Overview of Memory Protection

Many processors implement two levels of privilege, often called kernel and user, so that the most privileged code need not depend on the correctness of less privileged code. The operating system kernel has access to the entire processor, but disables access to certain features while application code runs to prevent the application from accessing or corrupting the kernel or other applications. This mechanism facilitates debugging and improves system reliability.

Some processors implement multiple levels of decreasing privilege, called rings, often with elaborate mechanisms for switching between rings. The Xtensa processor provides a configurable number of rings (`RingCount`), but without the elaborate ring-to-ring transition mechanisms. When configured with two rings, it provides the common kernel/user modes of operation, with Ring 0 being kernel and Ring 1 being user. With three or four rings configured, the Xtensa processor provides the same functionality as more advanced processors, but with the requirement that ring-to-ring transitions must be provided by Ring 0 (kernel) software.

Without the MMU Option, or with the MMU Option and `RingCount` = `1`, the Xtensa processor has a single level of privilege, and all instructions are always available.

With `RingCount` > `1`, software executing with `CRING` = `0` (see Table 4–63 on page 87 and the description of `PS.EXCM`) is able to execute all Xtensa instructions; other rings may only execute non-privileged instructions. The only distinction between the rings greater than zero is those created by software in the virtual-to-physical translations in the page table. The name “ring” is derived from an accessibility diagram for a single process such as that shown in Figure 4–19. At Ring 0 (that is, when `CRING` = `0`), the processor can access all of the current process’ pages (that is, Ring `0` to `RingCount-1` pages). At Ring 1 it can access all Ring `1` to `RingCount-1` pages. Thus, when the processor is executing with Ring 1 privileges, its address space is a subset of that at Ring 0 privilege, as Figure 4–19 illustrates. This concentric nesting of privilege levels continues to ring
`RingCount-1`, which can access only ring `RingCount-1` pages.

It is illegal for more than one TLB entry to match both the virtual address and the ASID. This is true even if the entries have different ASIDs which match at different ring levels. One ring’s mapping cannot not override another.

It is illegal for two or more TLB entries to match a virtual address, even if they are at different ring levels; one ring’s mapping cannot not override another.

Systems that require only traditional kernel/user privilege levels can, of course, configure `RingCount` to be 2. However, rings can also be useful for sharing. Many operating systems implement the notion of multiple threads sharing an address space, except for a small number of per-thread pages. Such a system could use Ring 0 for the shared kernel address space, Ring 1 for per-process kernel address space, Ring 2 for shared application address space, and Ring 3 for per-thread application address space.

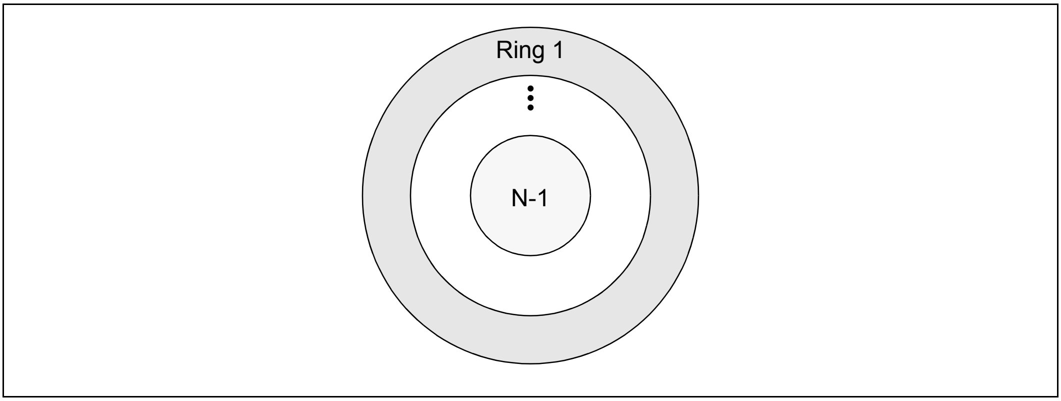


Figure 4–19. A Single Process' Rings

Each Xtensa ring has its own ASID. Ring 0's ASID is hardwired to 1. The ASIDs for Rings `1` to `RingCount-1` are specified in the `RASID` register. The ASIDs for each ring in `RASID` must be different. Each ASID has a single ring level, though there may be many ASIDs at the same ring level (except Ring 0). This allows nested privileges with sharing such as shown in Figure 4–20. The ring number of a page is not stored in the TLB; only the ASID is stored. When a TLB is searched for a virtual address match, the ASIDs of all rings specified in `RASID` are tried. The position of the matching ASID in RASID gives the ring number of the page. If the page's ring number is less than the processor's current ring number (`CRING`), then the access is denied with an exception (either `InstFetchPrivilegeCause` or `LoadStorePrivilegeCause`, as appropriate).

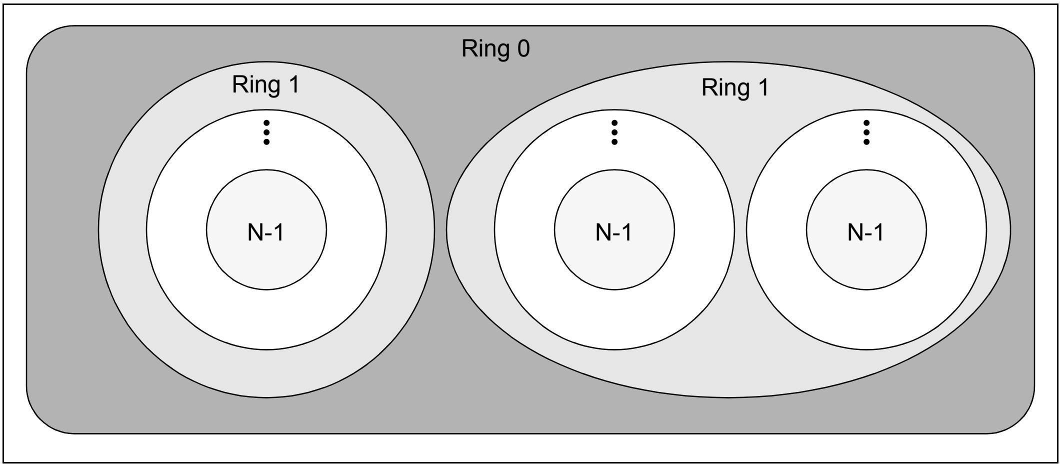


Figure 4–20. Nested Rings of Multiple Processes with Some Sharing

Why not store the ring number of the page in the TLB, and then use a single ASID for all rings, instead of having an ASID per ring? Because the latter allows sharing of TLB entries, and the former does not. For example, it is desirable at the very least to reuse the same TLB entries for all kernel mapped addresses, instead of having the same PTEs loaded into the TLB with different ASIDs. The Xtensa mechanism is more general than adding a "global" bit to each entry (to ignore the ASID match) in that it allows finer granularity, as Figure 4–20 illustrates, not just all or nothing.

The kernel typically assigns ASIDs dynamically as it runs code in different address spaces. When no more ASIDs are available for a new address space, the kernel flushes the Instruction and Data TLBs, and begins assigning ASIDs anew. For example, with `ASIDBits` = 8 and `RingCount` = 2, a TLB flush need occur at most every 254 context switches, if every context switch is to a new address space.

Note that `CRING` = 0 is the only requirement for privileged instructions to execute and `CRING` is the only field that controls access to memory. The `PS.UM` bit is named User Vector Mode and has nothing to do with privilege for either instructions or memory access. It controls only which exception vector is taken for general exceptions.

4.6.1.3 Overview of Attributes

Both page table entries (PTEs) and TLB entries store attribute bits that control whether and how the processor accesses memory. The number of potential attributes required by systems is large; to encode all the access capabilities required by any potential system would make this field too big to fit into a 4-byte PTE. However, the subset of values required for any particular system is usually much smaller. Each memory protection and translation option has a set of attributes, each of which encodes a set of capabilities from Table 4–99 for loads along with a set for stores and a set for instruction fetches. More capabilities are likely to be added in future implementations.

Table 4–99. Access Characteristics Encoded in the Attributes

Characteristic	Description	Used by
`Invalid`	Exception on access	Fetch, Load, Store
`Isolate`	Read/write cache contents regardless of tag compare	Load, Store
`Bypass`	Ignore cache contents regardless of tag compare — always access memory for this page	Fetch, Load, Store
`No-allocate`	Do not refill cache on miss	Fetch, Load, Store
`Write-through`	Write memory in addition to DataCache	Store
`Guarded`	Access bytes on this page exactly when required by the program (i.e. neither speculative references to reduce latency nor multiple accesses are allowed).	Load[1]

[1] Instruction fetch is always non-guarded. Stores are always guarded.

The assignment of capabilities to the attribute field of PTEs may be done with only one encoding for each distinct set of capabilities, or in such a way that each characteristic has its own bit, or anything in between. Often, single bits are used for a valid bit and a write-enable. For a valid bit, all of the attribute values with this bit zero would specify the Invalid characteristic so that any access causes an `InstFetchProhibitedCause`, `LoadProhibitedCause`, or `StoreProhibitedCause` exception, depending on the type of access. Similarly for the write-enable bit, all attribute values with write-enable zero would specify the Invalid characteristic to cause a `StoreProhibitedCause` exception on any store.

For systems that implement demand paging, software requires a page dirty bit to indicate that the page has been modified and must be written back to disk if it is replaced. This may be provided by creating a write-enable bit as described above, and using it as the per-page dirty bit. The first write to a clean (non-dirty) page causes a `StoreProhibitedCause` exception. The exception handler checks one of the software bits, which indicates whether the page is really writable or not; if it is, it then sets the hardware write-enable bit in both the TLB and the page table, and continues execution.

4.6.2 The Memory Access Process

All accesses to memory, whether to cache, local memories, XLMI, or PIF and whether caused by instruction fetch, the instructions themselves, or hardware TLB refill, follow certain steps. Following is a short description of these steps; each is discussed in more detail in Section 4.6.2.1 through Section 4.6.2.6.

1. **Choose the TLB:** Determine from the instruction opcode or the reason for hardware access, which TLB if any, is used for the access (see Section 4.6.2.1 on page 146 for details).
2. **Lookup in the TLB:** In that TLB, find an entry whose virtual page number matches the upper bits of the virtual address of the access and, for appropriate options, whose `ASID` matches one of the entries in the `RASID` register. Exactly one match is needed to continue beyond this point, although exceptions may be handled and the memory access process restarted (see Section 4.6.2.2 on page 147 for details).
3. **Check the access rights:** If the attribute is invalid or, for appropriate options, if the ring corresponding the `ASID` matched in the `RASID` register is too low, raise an exception. The operating system may, among other choices, modify the TLB entries and retry the access (see Section 4.6.2.3 on page 148 for details).
4. **Direct the access to local memory:** If the physical address of the access matches an instruction RAM or ROM, a data RAM or ROM, or an XLMI port then direct the access to that local memory or XLMI. An exception is possible at this stage for certain conditions, such as attempting to write to a ROM (see Section 4.6.2.4 on page 148 for details).

5. **Direct the access to PIF:** For the given cache configuration and using the attribute, determine whether to execute the required access on the processor interface bus (PIF) and make that access if necessary (see Section 4.6.2.5 on page 150 for details).
6. **Direct the access to cache:** Using the cache that corresponds to the TLB in Step 1 above, look up the memory location in the cache, using the value if it is there. If not, fill the cache from the PIF and then do the access (see Section 4.6.2.6 on page 150 for details).

Logically, the steps are done in order. The TLB lookup is done first (in steps 1 through 3 above) and the memory access afterwards (in steps 4 through 6 above). For performance reasons, they are actually done in parallel. This has two consequences:

1. First, the virtual and physical addresses of an access to an XLMI port must be identical so that the full address can be provided at the desired time.
2. Second, for all other local memory accesses and cacheable addresses, the index bits of the cache or local memory must be the same in both virtual and physical address. This means that caches which contain ways larger than the smallest page size in the system require “page coloring” as described in Section 4.6.1.1 on page 139.

For local memories, the second consequence requires a similar restriction on how they can be mapped. Note that local memories do not require that sequential virtual pages be mapped to sequential physical pages, but only that each virtual page be mapped to a physical page with which it shares the values of index bits.

For the purposes of understanding exceptions raised by memory accesses, all the steps above are done sequentially and the first exception encountered takes priority over later ones. For performance reasons, again, all steps are done in parallel and the results prioritized afterward.

The above steps are further expanded in the following subsections.

4.6.2.1 Choose the TLB

Several instructions do not actually address memory. They simply use the bits of an address to access a cache and do something directly to it. The following groups of instructions have this property:

- `III`, `IIU`
- `DII`, `DIU`, `DIWB`, `DIWBI`
- `LICT`, `SICT`, `LICW`, `SICW`
- `LDCT`, `SDCT`

For each of these instructions, no TLB is accessed and the remainder of the steps are not followed. No memory access exceptions are possible as the addresses are not really addresses but only pointers to cache locations.

For the data accesses of instructions IHI, IHU, IPF, and IPFL, as well as all instruction fetches, the instruction TLB is used for subsequent steps.

For the data accesses of all other instructions and for the hardware TLB refill accesses (regardless of which TLB is being refilled) the data TLB is used for subsequent steps.

The above choices are reflected in Table 4–100 in the second column.

For compatibility the two TLBs should never give conflicting translations or protection attributes for any access as future processors may implement them with only a single set of entries.

4.6.2.2 Lookup in the TLB

Each TLB lookup takes a virtual address as an operand and produces a physical address, a lookup ring, and attributes as a result. This process is described in more detail in Section 4.6.1.1. Each way of the TLB is read using the appropriate address bits for that way as index bits. For variable sized ways, the ITLBCFG or DTLBCFG register helps determine which address bits are the index bits.

For options without ASIDs (Region Protection Option), a way matches the access if its virtual page number (VPN) matches the VPN of the access. The lookup ring produced is defined to be 0.

For options with ASIDs (MMU Option), a way matches the access if its Virtual Page Number (VPN) matches the VPN of the access and the ASID of the way matches one of the ASIDs in the RASID register. The lookup ring is determined by which ASID in the RASID register is matched. Because the four entries in the RASID register are required to be different and non-zero, the lookup ring is well determined.

There should not be a match for more than one of the ways. However, this condition currently raises an InstTLBMultiHitCause or a LoadStoreTLBMultiHitCause exception as a debugging aid. If two entries contain the same VPN, but different ASIDs, they may co-exist in the TLB at the same time as long as the RASID never contains both ASIDs at the same time.

If none of the ways match, options without auto-refill ways (Region Protection Option) will raise an InstTLBMissCause or a LoadStoreTLBMissCause exception so that system software can take appropriate action and possibly retry the access. Options with auto-refill ways (MMU Option) will, automatically in hardware, use PTEVADDR to access page tables in memory and replace an entry in one of the auto-refill ways. The access will then be automatically retried. An error of any sort during the automatic refill process will raise an InstTLBMissCause or a LoadStoreTLBMissCause exception to be raised so that system software can take appropriate action and possibly retry the access.

If no exception is raised, the physical page number and attributes of the matching entry along with the lookup ring defined above are the results of the lookup and the access continues with the next step.

4.6.2.3 Check the Access Rights

First, the lookup ring of the entry is checked against the ring of the access. The ring of the access is usually CRING, but for L32E and S32E, for example, it is PS.RING instead. If the lookup ring of the entry is smaller than the ring of the access, an InstFetchPrivilegeCause or a LoadStorePrivilegeCause exception is raised. This situation means that an instruction has attempted access to a region of memory at a lower numbered ring than the one for which it has privilege.

Second, the attribute of the lookup is checked for validity. If the attribute is not valid, an exception is raised. If the access chose the Instruction TLB in Section 4.6.2.1, it raises an InstFetchProhibitedCause exception. If it chose the data TLB, it raises either a LoadProhibitedCause exception or a StoreProhibitedCause exception, depending on whether it was a load or a store.

If no exception is raised, the access continues with the next step using the physical address and the attribute (which is known to be valid for access, but may still affect how caches are used).

4.6.2.4 Direct the Access to Local Memory

The physical address of each access is compared to the address ranges of any instruction RAM, instruction ROM, data RAM, data ROM, or XLMI options that may exist in the processor. Table 4–100 indicates what will happen in the case that an access initiated by what is indicated in the Instruction column (which will use the TLB in the second column) if its address compares to an (abbreviated) option in one of the last six columns. OK means the access is completed normally. NOP means the access is completed but by its nature does nothing. IFE and LSE mean that an exception is raised. TLBI and TLBD mean that an InstTLBMissCause or a LoadStoreTLBMissCause exception is raised. Undef means the behavior is not defined.

Table 4–100. Local Memory Accesses

Instruction	TLB Used[1]	Inst-RAM	Inst-ROM	Data-RAM	Data-ROM	XLMI
Instruction-fetch	ITLB	OK	OK	IFE[2]	IFE[2]	IFE[2]
`IHI, IHU, IPF`	ITLB	NOP	NOP	NOP	NOP	NOP
`III, IIU`	none	—	—	—	—	—
`IPFL`	ITLB	IFE[5]	IFE[5]	IFE[2]	IFE[2]	IFE[2]
`L32I, L32R`	DTLB	OK[3]	OK[3]	OK	OK	OK
`L8UI, L16SI, L16UI, L32AI, L32E, FP Loads, MAC16 Loads`	DTLB	LSE[4]	LSE[4]	OK	OK	OK
`LICT, LICW, LDCT`	none	—	—	—	—	—
`S32I`	DTLB	OK[3]	LSE[4]	OK	LSE[4]	OK
`S8I, S16I, S32E, S32RI, FP Stores`	DTLB	LSE[4]	LSE[4]	OK	LSE[4]	OK
`S32C1I`	DTLB	LSE[4]	LSE[4]	OK[7]	LSE[4]	Undef
`SICT, SICW, SDCT`	none	—	—	—	—	—
`DHI, DHU, DHWB, DHWBI`	DTLB	NOP	NOP	NOP	NOP	NOP
`DII, DIU, DIWB, DIWBI`	none	—	—	—	—	—
`DPFR, DPFRO, DPFW, DPFWO`	DTLB	NOP	NOP	NOP	NOP	NOP
`DPFL`	DTLB	LSE[4]	LSE[4]	LSE[6]	LSE[6]	LSE[6]
`Hardware ITLB Refill`	DTLB	TLBI[8]	TLBI[8]	OK	OK	OK
`Hardware DTLB Refill`	DTLB	TLBD[8]	TLBD[8]	OK	OK	OK
`Designer defined loads`	DTLB	LSE[4]	LSE[4]	OK	OK	OK
`Designer defined stores`	DTLB	LSE[4]	LSE[4]	OK	LSE[4]	OK

[1] As described in Section 4.6.2.1 on page 146

[2] Raises exception - InstFetchErrorCause

[3] These accesses may be slow in some implementations.

[4] Raises exception - LoadStoreErrorCause

[5] Raises exception - InstFetchErrorCause - but not in all implementations

[6] Raises exception - LoadStoreErrorCause - but not in all implementations

[7] Works in newer implementations but in some older implementations raises an exception.

[8] Raises exception - InstTLBMissCause or a LoadStoreTLBMissCause depending on the original access.

Using the definition of guarded in Table 4–99, instruction-fetch accesses are never guarded. Stores are always guarded. Loads to instruction RAM, instruction ROM, data RAM, and data ROM are never guarded. These ports are assumed to be connected only to devices with memory semantics so that no guarding is needed for loads. Loads to

XLMI are only guarded in the sense that the load will be retired only under the conditions for a guarded access. For all these memories, assertion of the memory enable is no guarantee that the load was needed.

If none of the comparisons produces a match, the access continues with the next step using the physical address and the attribute.

4.6.2.5 Direct the Access to PIF

The access is sent to the processor interface if any of the following is true:

- The attribute indicates that the cache should be bypassed.
- The chosen TLB in Section 4.6.2.1 and in Table 4–100 is the ITLB and the Instruction Cache Option is not configured.
- The chosen TLB in Section 4.6.2.1 and in Table 4–100 is the DTLB and the Data Cache Option is not configured.

Using the definition of guarded in Table 4–99 on page 144, instruction-fetch accesses to the PIF are never guarded. Stores to the PIF are always guarded. Loads that are sent to the PIF under this section (without being cached) are guarded if the attribute says that they should be.

If the conditions of this section are not met, the access is cached and continues with the next step using the physical address and the attribute.

4.6.2.6 Direct the Access to Cache

The access is cached. The attribute determines how the cache operates, including the possibility of a write-through to the PIF.

The concept of guarding cannot be carried out for loads through the cache. Extra bytes have been loaded simply to fill the cache line and the line may have been filled long before the access. Inherently, the line is filled a different number of times than an access is executed and the line may be invalidated or evicted at any time and refilled later. Caching should not be used on ranges of memory address where guarding is important.

4.6.3 Region Protection Option

The simplest of the options, the Region Protection Option, provides a protection field for each of the eight 512 MB regions in the address space. The field can allow access to the region and it can set caching characteristics for the region, such as whether or not the cache is used and if it is write-through or write-back.

- Prerequisites: Exception Option (page 82)

- Incompatible options: MMU Option (page 158)

This simple option is built from the capabilities discussed in the introduction (Section 4.6.1). It uses `RingCount = 1`, so the processor can always execute privileged instructions. It sets `ASIDBits` to `0`, which disables the ASID feature. The instruction and data TLBs are programmed to each have one way of eight entries, and the VPNs (virtual page numbers) and PPNs (physical page numbers) of these entries are constant and hardwired to the identity map (that is, PPN = VPN). Only the attributes are not constant; they are writable using the `WITLB` and `WDTLB` instructions.

4.6.3.1 Region Protection Option Architectural Additions

Table 4–101 through Table 4–103 show this option's architectural additions.

Table 4–101. Region Protection Option Exception Additions

Exception	Description	EXCCAUSE value
InstFetchProhibitedCause	Instruction fetch is not allowed in region	20
LoadProhibitedCause	Load is not allowed in region	28
StoreProhibitedCause	Store is not allowed in region	29

Table 4–102. Region Protection Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Access
ITLB Entries	8	4	Instruction TLB entries	R/W	see Table 4–103
DTLB Entries	8	4	Data TLB entries	R/W	see Table 4–103

Table 4–103. Region Protection Option Instruction Additions

Instruction[1]	Format	Definition
IDTLB	RRR	Invalidate data TLB entry
IITLB	RRR	Invalidate instruction TLB entry
PDTLB	RRR	Probe data TLB
PITLB	RRR	Probe instruction TLB
RDTLB0	RRR	Read data TLB virtual
RDTLB1	RRR	Read data TLB translation
RITLB0	RRR	Read instruction TLB virtual

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

Table 4–103. Region Protection Option Instruction Additions (continued)

Instruction[1]	Format	Definition
`RITLB1`	RRR	Read instruction TLB translation
`WDTLB`	RRR	Write data TLB
`WITLB`	RRR	Write instruction TLB

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.6.3.2 Formats for Accessing Region Protection Option TLB Entries

During normal operation when instructions and data are being accessed from memory, only lookups are being done in the TLBs. For maintenance of the TLBs, however, the entries in the TLBs are accessed by the instructions in Table 4–103. Note that unused bits at Bit 12 and above are ignored on write, and zero on read, so that those bits may simply contain the address for access to all ways of both TLBs. Unused bits at Bit 11 and below are required to be zero on write and undefined on read for forward compatibility.

The format of the `as` register used in all instructions in the table is shown in Figure 4–21. The upper three bits are used as an index among the TLB entries just as they would be when addressing memory. They are the Virtual Page Number (VPN) or upper three bits of address. The remaining bits are ignored.

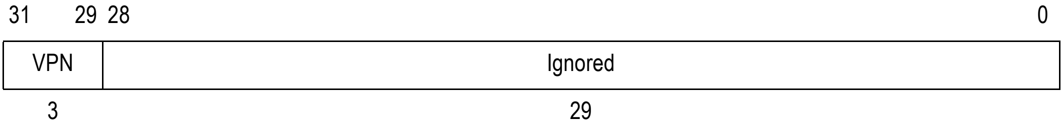


Figure 4–21. Region Protection Option Addressing (`as`) Format for `WxTLB`, `RxTLB1`, & `PxTLB`

The `WITLB` and `WDTLB` instructions write the TLB entries. The `as` register is formatted according to Figure 4–21, while the `at` register is formatted according to Figure 4–22. The attribute for the region is described in detail in Section 4.6.3.3. The remaining bits are ignored or required to be zero.

After modifying any TLB entry with a `WITLB` instruction, an `ISYNC` must be executed before executing any instruction from that region. In the special case of the `WITLB` changing the attribute of its own region, the `ISYNC` must immediately follow the `WITLB` and both must be within the same memory region and, if the region is cacheable, within the same cache line.

31 12	11 4	3 0
Ignored	Zero	Attribute
20	8	4

Figure 4–22. Region Protection Option Data (`at`) Format for `WxTLB`

The `RITLB0` and `RDTLB0` instructions exist under this option but do not return interesting information because the entire VPN is used as an index. The `as` register is formatted according to Figure 4–21. The read instructions return zero in the `at` register.

The `RITLB1` and `RDTLB1` instructions return the `at` data format in Figure 4–23. The Attribute for the region is described in detail in Section 4.6.3.3. The VPN is returned in the upper three bits as the Physical Page Number (PPN) because there is no translation. The remaining bits are zero or undefined. The `as` register is formatted according to Figure 4–21.

31 29	28 12	11 4	3 0
PPN	Zero	Undefined	Attribute
3	17	8	4

Figure 4–23. Region Protection Option Data (`at`) Format for `RxTLB1`

The `PITLB` and `PDTLB` instructions exist under this option but do not return interesting information because all accesses hit in the respective TLBs and the TLBs have only a single way. The `as` register is formatted according to Figure 4–21. The TLB probe instructions return the `at` data format in Figure 4–24. The VPN is returned in the upper bits. The low bit is set because the probe always hits and the remaining bits are zero or undefined.

31 29	28 12	11 1	0
VPN	Zero	Undefined	1
3	17	11	1

Figure 4–24. Region Protection Option Data (`at`) Format for `PxTLB`

The `IITLB` and `IDTLB` instructions exist under this option and their `as` register is formatted according to Figure 4–21, but they have no effect because the entries cannot be removed from the respective TLBs.

4.6.3.3 Region Protection Option Memory Attributes

The memory attributes written into the TLB entries by the `WxTLB` instructions and read from them by the `RxTLB1` instructions control access to memory and, where there is a cache, how the cache is used. Table 4–104 shows the meanings of the attributes for instruction fetch, data load, and data store. For a more detailed description of the memory access process and the place of these attributes in it, see Section 4.6.2.

The first column in Table 4–104 indicates the attribute attribute from the TLB while the remaining columns indicate various effects on the access. The columns are described in the following bullets:

- **Attr** — the value of the 4-bit Attribute field of the TLB entry.
- **Rights** — whether the TLB entry may successfully translate a data load, a data store, or an instruction fetch.
  - The first character is an `r` if the entry is valid for a data load and a dash ("`-`") if not.
  - The second character is a `w` if the entry is valid for a data store and a dash ("`-`") if not.
  - The third character is an `x` if the entry is valid for an instruction fetch and a dash ("`-`") if not.

  If the translation is not successful, an exception is raised.

  Local memory accesses (including XLMI) consult only the Rights column.
- **WB** — some rows are split by whether or not the configured cache is writeback or not. Rows without an entry apply to both cache types.
- **Meaning for Cache Access** — the verbal description of the type of access made to the cache.
- **Access Cache** — indicates whether the cache provides the data.
  - The first character is an `h` if the cache provides the data when the tag indicates hit and a dash ("`-`") if it does not.
  - The second character is an `m` if the cache provides the data when the tag indicates a miss and a dash ("`-`") if it does not. This capability is used only for Isolate mode.
- **Fill Cache** — indicates whether an allocate and fill is done to the cache if the tag indicates a miss.
  - The first character is an `r` if the cache is filled on a data load and a dash ("`-`") if it is not.
  - The second character is a `w` if the cache is filled on a data store and a dash ("`-`") if it is not.
  - The third character is an `x` if the cache is filled on an instruction fetch and a dash ("`-`") if it is not.

- **Guard Load** — refers to the guarded attribute as described in Table 4–99 on page 144. Stores are always guarded and instruction fetches are never guarded, but loads are guarded where there is a "yes" in this column. Local memory loads are not guarded.
- **Write Thru** — indicates whether a write is done through the PIF interface.
  - The first character is an `h` if a Write Thru occurs when the tag indicates hit and a dash ("-") if it does not.
  - The second character is an `m` if a Write Thru occurs when the tag indicates a miss and a dash ("-") if it does not.

Writes to local memories are never Write-Thru. In most implementations, a write-thru will only occur after any needed cache fill is complete.

Table 4–104. Region Protection Option Attribute Field Values

Attr	Rights	Meaning for Cache Access	Access Cache	Fill Cache	Guard Load	Write Thru
0	rw-	Cached, No Allocate	h-	---	-	hm
1	rwx	Cached, WrtThru	h-	r-x	-	hm
2	rwx	Bypass cache	--	---	yes	hm
3	--x	Cached[1]	h-	--x	-	--
4	rwx	Cached, WrtBack alloc	h-	rwx	-	--
5	rwx	Cached, WrtBack noalloc[1]	h-	r-x	-	-m
6-13	---	Reserved[2]	—	—	—	—
14	rw-	Cache Isolated[3]	hm	---	-	--
15	---	illegal[2]	--	---	-	--

[1] Attribute not supported in all implementations. Please refer to a specific Xtensa processor data book for supported attributes.

[2] Raises exception. EXCCAUSE is set to InstFetchProhibitedCause, LoadProhibitedCause, or StoreProhibitedCause depending on access type

[3] For test only, implementation dependent, uses data cache like local memories and ignores tag.

All attribute entries in the ITLB and DTLB are set to cache bypass (4'h2) after reset.

In the absence of the Instruction Cache Option, Cached regions behave as Bypass regions on instruction fetch. In the absence of the Data Cache Option, Cached regions behave as Bypass regions on data load or store. If the Data Cache is not configured as writeback (Section 4.5.5.1 on page 119) Attributes 4 and 5 behave as Attribute 1 instead of as they are listed in Table 4–104.

After changing the attribute of any memory region with a `WITLB` instruction, an `ISYNC` must be executed before executing any instruction from that region. In the special case of the `WITLB` changing the attribute of its own region, the `ISYNC` must immediately follow the `WITLB` and both must be within the same cache line.

After changing the attribute of a region by WDTLB, the operation of loads from and stores to that region are undefined until a DSYNC instruction is executed.

4.6.4 Region Translation Option

Building on the Region Protection Option is the Region Translation Option, which adds a virtual-to-physical translation on the upper three bits of the address. Thus, each of the eight 512 MB regions, in addition to the attributes provided by the Region Protection Option, may be redirected to access a different region of physical address space.

- Prerequisites: Exception Option (page 82) and Region Protection Option (page 150)
- Incompatible options: MMU Option (page 158)

With this option, the Physical Page Numbers (PPNs) of each of the TLB entries is now writable instead of constant and identity mapped. In this way, the same region of memory may be accessed with different attributes by the use of different virtual addresses.

This simple option is built from the capabilities discussed in the introduction (see Section 4.6.1). It uses RingCount = 1, so the processor can always execute privileged instructions. It sets ASIDBits to 0, which disables the ASID feature. The instruction and data TLBs are programmed to each have one way of eight entries, and only the attributes and Physical Page Numbers (PPNs) are not constant; they are writable using the WITLB and WDTLB instructions.

4.6.4.1 Region Translation Option Architectural Additions

There are no new exceptions, no new state registers, and no new Instructions added to those in the Region Protection Option. The TLB entries contain three additional bits of state. Access to these bits is described in Section 4.6.4.2.

4.6.4.2 Region Translation Option Formats for Accessing TLB Entries

During normal operation when instructions and data are being accessed from memory, only lookups are being done in the TLBs. For maintenance of the TLBs, however, the entries in the TLBs are accessed by the instructions in Table 4–103 on page 151. Note that unused bits at Bit 12 and above are ignored on write and zero on read so that those bits may simply contain the address for access to all ways of both TLBs. Unused bits at Bit 11 and below are required to be zero on write and undefined on read for forward compatibility.

The register formats used by the TLB instructions are very similar to those described in Section 4.6.3.2 for the Region Protection Option. The only difference is the presence of a Physical Page Number (PPN) in the upper three bits of the WxTLB, RxTLB1, and PxTLB register formats.

The format of the as register used in all instructions in the table is shown in Figure 4–25. The upper three bits are used as an index among the TLB entries just as they would be when addressing memory. They are the Virtual Page Number (VPN) or upper three bits of address. The remaining bits are ignored.

31 29	28 0
VPN	Ignored
3	29

Figure 4–25. Region Translation Option Addressing (as) Format for WxTLB, RxTLB1, & PxTLB

The WITLB and WDTLB instructions write the TLB entries. The as register is formatted according to Figure 4–25, while the at register is formatted according to Figure 4–26. The attribute for the region is described in detail in Section 4.6.3.3 on page 154. The remaining bits are ignored or required to be zero.

After modifying any TLB entry with a WITLB instruction, an ISYNC must be executed before executing any instruction from that region. In the special case of the WITLB changing the attribute of its own region, the ISYNC must immediately follow the WITLB and both must be within the same memory region and, if the region is cacheable, within the same cache line.

After modifying any TLB entry with a WDTLB instruction, the operation of loads from and stores to that region are undefined until a DSYNC instruction is executed.

31	12	11 4	3 0
PPN	Ignored	Zero	Attribute
3	17	8	4

Figure 4–26. Region Translation Option Data (at) Format for WxTLB

The RITLB0 and RDTLB0 instructions exist under this option but do not return interesting information because the entire VPN is used as an index. The as register is formatted according to Figure 4–25. The read instructions return zero in the at register.

The RITLB1 and RDTLB1 instructions return the at data format in Figure 4–27. The attribute for the region is described in detail in Section 4.6.3.3. The Physical Page Number (PPN) is returned in the upper three bits. The remaining bits are zero or undefined. The as register is formatted according to Figure 4–25.

31 29	28 12	11 4	3 0
PPN	Zero	Undefined	Attribute
3	17	8	4

Figure 4–27. Region Translation Option Data (`at`) Format for `RxTLB1`

The `PITLB` and `PDTLB` instructions return the `at` data format in Figure 4–28. The Virtual Page Number (VPN) is returned in the upper bits. The low bit is set because the probe always hits, and the remaining bits are zero or undefined. The `as` register is formatted according to Figure 4–25. These instructions work for their intended purpose, but do not provide useful information under this simple option because the TLBs always hit and have only a single way.

31 29	28	1	0
VPN	Zero	Undefined	1
3	17	11	1

Figure 4–28. Region Translation Option Data (`at`) Format for `PxTLB`

The `IITLB` and `IDTLB` instructions exist under this option and their `as` register is formatted according to Figure 4–25, but they have no effect because the entries cannot be removed from the respective TLBs.

4.6.4.3 Region Translation Option Memory Attributes

The memory attributes written into the TLB entries by the `WxTLB` instructions and read from them by the `RxTLB1` instructions are exactly the same as under the Region Protection Option.

As with the Region Protection Option, all attributes in both TLBs are set to cache bypass (`4’b0010`) after reset. In addition, the translation entries in both TLBs are set to identity map after reset.

4.6.5 MMU Option

The MMU Option is a memory management unit created to run protected operating systems such as Linux on the Xtensa processor with demand paging hardware with a memory-based page table.

- Prerequisites: Exception Option (page 82)

- Incompatible options: Region Protection Option (page 150), Extended L32R Option (page 56)

This option is also built from the capabilities discussed in the introduction (Section 4.6.1). It uses RingCount = 4 and only Ring 0 may execute privileged instructions. The option sets ASIDBits to 8, which allows for lower TLB management overhead.

The instruction and data TLBs are programmed to have seven and ten ways, respectively (see Section 4.6.5.3). Some of the ways are constants; others can be set to arbitrary values. Still others auto-refill from a page table in memory that contains 4-byte PTEs, each mapping a 4kB page with a 20-bit PPN, a 2-bit ring number, a 4-bit attribute, and 6 bits reserved for software. For a programmer's view of the MMU, refer to the *Xtensa Microprocessor Programmer's Guide*.

4.6.5.1 MMU Option Architectural Additions

Table 4–105 through Table 4–108 show this option's architectural additions.

Table 4–105. MMU Option Processor-Configuration Additions

Parameter	Description	Valid Values
NIREFILLENTRIES	Number of auto-refill entries in the ITLB (divided among 4 ways)	16,32 (4, 8 entries per TLB way)
NDREFILLENTRIES	Number of auto-refill entries in the DTLB (divided among 4 ways)	16,32 (4, 8 entries per TLB way)
IVARWAY56	Ways 5&6 of the ITLB can be variable for greater flexibility in mapping memory	Variable or Fixed[1]
DVARWAY56	Ways 5&6 of the DTLB can be varialble for greater flexitiblity in mapping memory	Variable or Fixed[1]

[1] Implementations may allow only Fixed, only Variable or a choice of either for this value.

Table 4–106. MMU Option Exception Additions

Exception	Description	EXCCAUSE Value
PrivilegedCause	Privileged instruction attempted with CRING $\neq 0$	8
InstTLBMissCause	Instruction fetch finds no entry in ITLB	16
InstTLBMultiHitCause	Instruction fetch finds multiple entries in ITLB	17
InstFetchPrivilegeCause	Instruction fetch matching entry requires lower CRING	18
InstFetchProhibitedCause	Instruction fetch is not allowed in region	20
LoadStoreTLBMissCause	Load/store finds no entry in DTLB	24

Table 4–106. MMU Option Exception Additions (continued)

Exception	Description	EXCCAUSE Value
LoadStoreTLBMultiHitCause	Load/store finds multiple entries in DTLB	25
LoadStorePrivilegeCause	Load/store matching entry requires lower CRING	26
LoadProhibitedCause	Load is not allowed in region	28
StoreProhibitedCause	Store is not allowed in region	29

Table 4–107. MMU Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
PS.RING	1	2	Privilege level (see Table 4–63 on page 87)	R/W	230
PTEVADDR	1	32	Page Table Virtual Address	R/W	83
RASID	1	32	Per-ring ASIDs	R/W	90
ITLBCFG	1	2/4	Instruction TLB configuration	R/W	91
DTLBCFG	1	2/4	Data TLB configuration	R/W	92
ITLB Entries	24,32,40,48[2]	variable	Instruction TLB entries	R/W	Table 4–108
DTLB Entries	27,35,43,51[2]	variable	Data TLB entries	R/W	Table 4–108

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 5–127 on page 205. The TLB Entries are not Special Registers, but are accessed by the instructions in Table 4–108 on page 160.

[2] See Section 4.6.5.3 on page 163 for more information on TLB structure.

Table 4–108. MMU Option Instruction Additions

Instruction[1]	Format	Definition
IDTLB	RRR	Invalidate data TLB entry
IITLB	RRR	Invalidate instruction TLB entry
PDTLB	RRR	Probe data TLB
PITLB	RRR	Probe instruction TLB
RDTLB0	RRR	Read data TLB virtual
RDTLB1	RRR	Read data TLB Translation
RITLB0	RRR	Read instruction TLB virtual

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

Table 4–108. MMU Option Instruction Additions (continued)

Instruction[1]	Format	Definition
`RITLB1`	RRR	Read instruction TLB translation
`WDTLB`	RRR	Write data TLB
`WITLB`	RRR	Write instruction TLB

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

4.6.5.2 MMU Option Register Formats

This section describes the address and data formats needed for reading and writing the instruction and data TLBs.

PTEVADDR

Because four ways of each TLB are configured as AutoRefill, the MMU Option supports hardware refill of the TLB from a page table (Section 4.6.5.9). The base virtual address of the current page table is specified in the PTEBase field of the `PTEVADDR` register. When read, `PTEVADDR` returns the PTEBase field in its upper bits as shown in Figure 4–29, `EXCVADDR`$_{31..12}$ in the field labeled `VPN` below followed by two zero bits. When `PTEVADDR` is written, only the PTEBase field is modified. `PTEVADDR` is undefined after reset. Figure 4–29 shows the `PTEVADDR` register format.

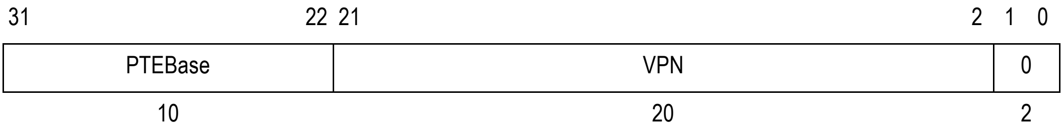


Figure 4–29. MMU Option PTEVADDR Register Format

RASID

The Ring ASID (`RASID`) register holds the current ASIDs for each ring. The register is divided into four 8-bit sections, one for each ASID. The Ring 0 ASID is hardwired to `1`. The operation of the processor is undefined if any two of the `four` ASIDs are equal or if it contains an ASID of zero. `RASID` is `32'h04030201` after reset. Figure 4–30 shows the `RASID` register format.

31 24	23 16	15 8	7 0
Ring3 ASID	Ring2 ASID	Ring1 ASID	8'h01
8	8	8	8

Figure 4–30. MMU Option RASID Register Format

ITLBCFG

Because one or three ways of the instruction TLB are configured with variable page sizes (depending on whether `IVARWAY56` is, respectively, fixed or variable), the `ITLBCFG` register specifies the page size for those ways. Regardless of `IVARWAY56`, the Size field in bits[17:16] of the register controls the size of the entries in Way 4 and has the values 2'b00 = 1 MB, 2'b01 = 4 MB, 2'b10 = 16 MB, and 2'b11 = 64 MB. If `IVARWAY56` is Variable, the Sz field in bit[20] of the register controls the size of the entries in Way 5 and has the values 1'b0 = 128MB and 1'b1 = 256MB. If `IVARWAY56` is Variable, the Sz field in bit[24] of the register controls the size of the entries in Way 6 and has the values 1'b0 = 512MB and 1'b1 = 256MB. MBZ means “must be zero”. The entire TLB way should be invalidated when its size is changed. The `ITLBCFG` register is zero after reset. The following shows the `ITLBCFG` register format.

31 25	24	23 21	20	19 18	17 16	15 0
MBZ	Sz	MBZ	Sz	MBZ	Size	MBZ
7	1	3	1	2	2	16

MMU Option ITLBCFG Register Format

DTLBCFG

Because one or three ways of the data TLB are configured with variable page sizes (depending on whether `DVARWAY56` is, respectively, fixed or variable), the `DTLBCFG` register specifies the page size for those ways. Regardless of `DVARWAY56`, the Size field in bits[17:16] of the register controls the size of the entries in Way 4 and has the values 2'b00 = 1 MB, 2'b01 = 4 MB, 2'b10 = 16 MB, and 2'b11 = 64 MB. If `DVARWAY56` is Variable, the Sz field in bit[20] of the register controls the size of the entries in Way 5 and has the values 1'b0 = 128MB and 1'b1 = 256MB. If `DVARWAY56` is Variable, the Sz field in bit[24] of the register controls the size of the entries in Way 6 and has the values 1'b0 = 512MB and 1'b1 = 256MB. MBZ means “must be zero”. The entire TLB way should be invalidated when its size is changed. The `DTLBCFG` register is zero after reset.
Figure 4–31 shows the `DTLBCFG` register format.

31 25	24	23 21	20	19 18	17 16	15 0
MBZ	Sz	MBZ	Sz	MBZ	Size	MBZ
7	1	3	1	2	2	16

Figure 4–31. MMU Option DTLBCFG Register Format

4.6.5.3 The Structure of the MMU Option TLBs

The instruction TLB is 7-way set-associative. Ways 0-3 are AutoRefill ways used for hardware refill of 4 kB page table entries from the page table when no matching TLB entry is found. The AutoRefill ways contain a total of either 16 entries (four per way) or 32 entries (eight per way) depending on `NIREFILLENTRIES`. Way 4 is a variable size way of four entries and is used for mapping large pages of 1 MB, 4 MB, 16 MB, or 64 MB as configured by the `ITLBCFG` register. The `ASID` fields in these ways are set to zero (invalid) after reset.

Way 5 (`IVARWAY56` Fixed), with two constant entries, statically maps the 128 MB region `32'hD0000000-32'hD7FFFFFF` to the first 128 MB of physical memory (`32'h00000000-32'h07FFFFFF`) as cached memory (attribute 4’h7 as described in Section 4.6.5.10), and the next 128 MB region (`32'hD8000000-32'hDFFFFFFF`) to the same 128 MB of physical memory as cache bypassed memory (attribute 4’h3 as described in Section 4.6.5.10). The `ASID` entries for both entries is 8’h01. These 128 MB regions are intended for the operating system kernel’s first 128 MB of code and data (see Figure 4–32). Using a pair of large static mappings reduces the load on the demand refill portion of the instruction TLB and also provides access using two attributes for the same memory. Physical memory above the first 128 MB is accessed via dynamically mapped virtual address space.

Way 5 (`IVARWAY56` Variable), is a variable size way of four entries and is used for mapping very large pages of 128 MB or 256 MB as configured by the `ITLBCFG` register. The `ASID` fields in this way are set to zero (invalid) after reset. This way may be used to emulateWay 5 (`IVARWAY56` Fixed), or it may be used for a more flexible arrangement.

Way 6 (`IVARWAY56` Fixed), also with 2 constant entries, statically maps the 256 MB region `32'hE0000000-32'hEFFFFFFF` to the last 256 MB of physical memory (`32'hF0000000-32'hFFFFFFFF`) as cached memory (attribute 4’h7 as described in Section 4.6.5.10), and the next 256 MB region (`32'hF0000000-32'hFFFFFFFF`) to the same 256MB of physical memory as cache bypassed memory (attribute 4’h3 as described in Section 4.6.5.10). The `ASID` entries for both entries is 8’h01. These 256 MB regions are intended for addressing the system peripherals (for example, a PCI or other I/O bus) and system ROM (see Figure 4–32).

Way 6 (IVARWAY56 Variable), is a variable size way of eight entries and is used for mapping very large pages of 512 MB or 256 MB as configured by the ITLBCFG register. The ASID fields in this way are set one and the Attribute fields in this way are set to 4’h2 (Bypass) after reset, and the other fields are set so that this way directly maps all of memory after reset. This way may be used to emulate Way 6 (IVARWAY56 Fixed), it may be used to effectively "turn off" the ITLB, or it may be used for a more flexible arrangement.

The data TLB is 10-way set-associative. It has the same seven ways as the instruction TLB above (using DTLBCFG/DVARWAY56, instead of ITLBCFG/IVARWAY56), with the addition of Ways 7-9, which are single-entry ways for 4 kB pages. These ways are intended to hold translations required to map the page table for hardware refill and for entries that are not to be replaced by refill. The ASID fields in these ways are set to zero (invalid) after reset.

All ASID fields in the ITLB and DTLB, except those in Way 5 & Way 6, are set to zero (invalid) after reset. ASID fields in Way 5 are set to zero (invalid) after reset if IVARWAY56/DVARWAY56 is Variable.

4.6.5.4 The MMU Option Memory Map

The memory map is determined by the TLB configurations given in Section 4.6.5.3. Figure 4–32 shows a graphical representation of the constant translations in Way 5 and Way 6 when IVARWAY56 and DVARWAY56 are Fixed, as well as the regions that are mapped by more flexible ways than these. Way 5 and Way 6 may be used to emulate this same arrangement when IVARWAY56 and DVARWAY56 are Variable.

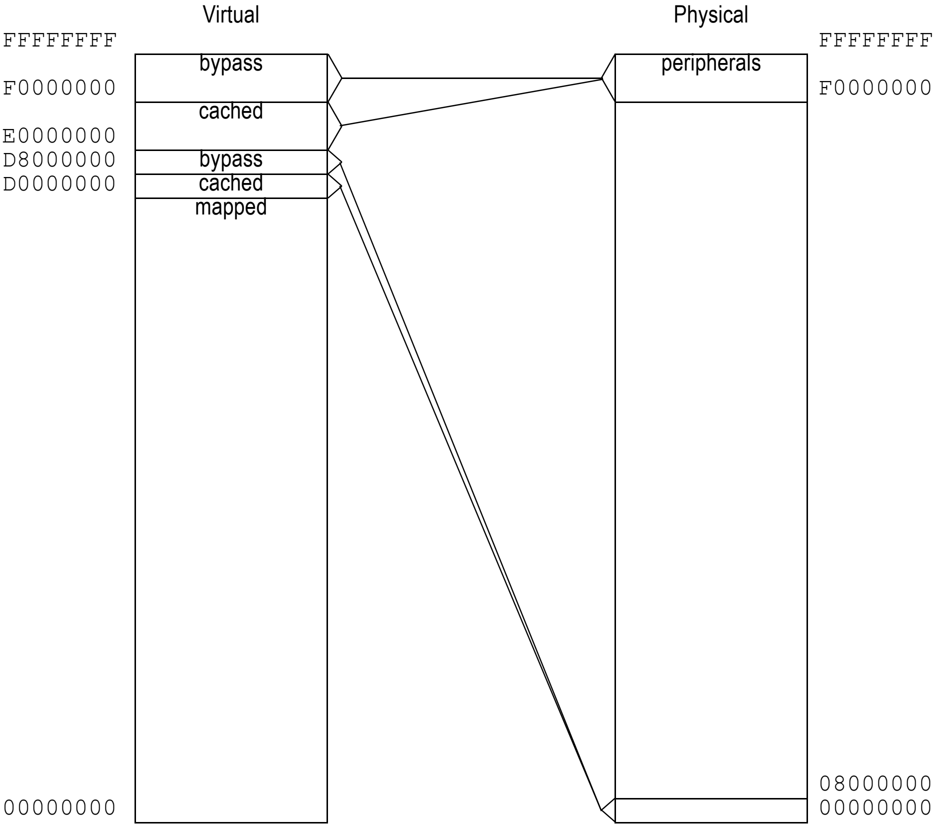


Figure 4–32. MMU Option Address Map with `IVARWAY56` and `DVARWAY56` Fixed

This configuration provides both bypass and cached access to peripherals. Bypass access is used for devices and cached access is used for ROMs, for example. It also provides bypass and cached access to the low 128 MB of memory. This allows system software to access its memory without competing with user code for other TLB entries. These are available after reset. The large page way (Way 4) and the auto-refill ways (Ways 0-3) may be used to map as much additional space as desired (Section 4.6.5.9). In the data TLB, Ways 7-9 may be used to map single pages so that they are always available.

4.6.5.5 Formats for Writing MMU Option TLB Entries

During normal operation when instructions and data are being accessed from memory, only lookups are being done in the TLBs. For maintenance of the TLBs, however, the entries in the TLBs are accessed by the instructions in Table 4–108 on page 160.

Writing the TLB with the WITLB and WDTLB instructions requires the formats for the as and at registers shown in Figure 4–33 and Figure 4–34. These figures show, in parallel, the formats for different ways of the cache and different conditions. For Ways 0-3, there are two conditions that depend on the configuration parameter NIREFILLENTRIES or NDREFILLENTRIES (see Figure 4–105 on page 159) and can have the values of 16 or 32 auto-refill entries per TLB (four or eight per TLB way). For Way 4, there are four conditions, which are the four values of the respectiveITLBCFG or DTLBCFG fields and indicate the size of pages currently contained within that way. Ways 5 and 6 can be Fixed or Variable as determined by the IVARWAY56 and DVARWAY56 parameters. If they are variable then there are still two conditions which are the two values of the respective ITLB-CFG or DTLBCFG fields and indicate the size of pages currently contained within that way. Each row, then, contains the format for the way and condition indicated in the left column. Note that writing to Way-5 and Way-6 when the IVARWAY56 and DVARWAY56 parameters are "Fixed" causes no changes because those ways are constant.

Writing ITLB Ways 7-15 or DTLB ways 10-15 is undefined.

The format of the as register used for the WITLB and WDTLB instructions is shown in Figure 4–33. The low order four bits contain the way to be accessed. The upper bits contain the Virtual Page Number (VPN). For clarity, the Index bits are separated out from the rest of the VPN in this figure. Note that unused bits at Bit 12 and above are ignored so that those bits may simply contain the address for access to all ways of both TLBs. Unused bits at Bit 11 and below are reserved for forward compatibility. They may either be zero or they may be the result of the probe instruction (Section 4.6.5.7).

Way 31 30 29 28 27 26 25 24 23 22 21 20 19 15 14 13 12 11 4 3 2 1 0

0-3 (16entry)	VPN without Index	Index		Reserved	4’h0,1,2,3
0-3 (32entry)	VPN without Index	Index		Reserved	4’h0,1,2,3
4 (1MB)	VPN without Index	Index	Ignored	Reserved	4’h4
4 (4MB)	VPN without Index	Index	Ignored	Reserved	4’h4
4 (16MB)	VPN without Index	Index	Ignored	Reserved	4’h4
4 (64MB)	VPN w/o Idx	Index	Ignored	Reserved	4’h4
5 (Fixed)	Ignored			Reserved	4’h5
5 (128MB)	VPN	Index	Ignored	Reserved	4’h5
5 (256MB)	VPN	Index	Ignored	Reserved	4’h5
6 (Fixed)	Ignored			Reserved	4’h6
6 (512MB)	Index	Ignored		Reserved	4’h6
6 (256MB)	V	Index	Ignored	Reserved	4’h6

Figure 4–33. MMU Option Addressing (as) Format for WxTLB

The format of the at register used for the WITLB and WDTLB instructions is shown in Figure 4–34. The low order four bits contain the attribute to be written (see Section 4.6.5.10). The two bits above those contain the ring for which this TLB entry is to be written. The ASID taken from the RASID register (see Section 4.6.5.2) corresponding to this ring is stored with the TLB entry. It is not possible to write an entry with an ASID which is not currently in the RASID register. The upper bits contain the Physical Page Number (PPN) of the translation. Way-5 and Way-6 are constant ways when the IVARWAY56 and DVARWAY56 parameters are "Fixed": The PPN remains as described in Section 4.6.5.3, the ASID is not written but always matches Ring 0, and the attribute remains as described in Section 4.6.5.3, no matter what is in register at. As with the address format, unused bits at Bit 12 and above are ignored so that a 20-bit PPN may be used with all ways of the TLB, and unused bits at Bit 11 and below are required to be zero for forward compatibility.

Way	31 29 28 27 26 25 24 23 22 21 20 19 18 17	12	11 6	5 4	3 0
0-3 (16entry)	PPN		6’h00	Ring	Attribute
0-3 (32entry)	PPN		6’h00	Ring	Attribute
4 (1MB)	PPN	Ignored	6’h00	Ring	Attribute
4 (4MB)	PPN	Ignored	6’h00	Ring	Attribute
4 (16MB)	PPN	Ignored	6’h00	Ring	Attrbute
4 (64MB)	PPN	Ignored	6’h00	Ring	Attribute
5 (Fixed)	Ignored		6’h00	Ignored	
5 (128MB)	PPN	Ignored	6’h00	Ring	Attribute
5 (256MB)	PPN	Ignored	6’h00	Ring	Attribute
6 (Fixed)	Ignored		6’h00	Ignored	
6 (512MB)	PPN	Ignored	6’h00	Ring	Attribute
6 (256MB)	PPN	Ignored	6’h00	Ring	Attribute
7-9(DTLB)	PPN		6’h00	Ring	Attribute
	31 29 28 27 26 25 24 23 22 21 20 19 18 17	12	11 6	5 4	3 0

Figure 4–34. MMU Option Data (`at`) Format for `WxTLB`

After modifying any TLB entry with a `WITLB` instruction, an `ISYNC` must be executed before executing any instruction that depends on the modification. The ITLB entry currently being used for instruction fetch may not be changed.

After modifying any TLB entry with a `WDTLB` instruction, the operation of loads and stores that depend on that TLB entry are undefined until a `DSYNC` instruction is executed.

4.6.5.6 Formats for Reading MMU Option TLB Entries

Reading the TLB with the `RITLB0`, `RITLB1`, `RDTLB0`, and `RDTLB1` instructions requires the formats for the `as` and `at` registers shown in Figure 4–35 through Figure 4–37. These figures show, in parallel, the formats for different ways of the cache and different conditions. For Ways 0-3, there are two conditions that depend on the configuration parameter `NIREFILLENTRIES` or `NDREFILLENTRIES` (see Figure 4–105 on page 159) and can have the values of 16 or 32 auto-refill entries per TLB (four or eight per TLB way). For Way 4, there are four conditions, which are the four values of the respective`ITLBCFG` or `DTLBCFG` fields and indicate the size of pages currently contained within that way. Ways 5 and 6 can be Fixed or Variable as determined by the `IVARWAY56` and `DVARWAY56` parameters. If they are variable then there are still two conditions which are the two values of the respective `ITLBCFG` or `DTLBCFG` fields and indicate the size of pages currently contained within that way. Each row, then, contains the format for the way and condition indicated in the left column.

Reading ITLB ways 7-15 or DTLB ways 10-15 is undefined.

The format of the `as` register used for the `RITLB0`, `RITLB1`, `RDTLB0`, and `RDTLB1` instructions is shown in Figure 4–35. The low order four bits contain the way to be accessed. Besides the Way bits, only the Index bits are needed for reading the TLB. Depending on the TLB way being accessed, and other conditions such as the size assigned to the variable size way or the number of auto refill entries in the TLB, different bits of address may be needed as shown. Note that unused bits at Bit 12 and above are ignored so that an entire 20-bit VPN may be used when accessing all ways of both TLBs. Unused bits at Bit 11 and below are reserved for forward compatibility. They may either be zero or they may be the result of the probe instruction (Section 4.6.5.7).

<table>
<tr><th>Way</th><th>31</th><th></th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th></th><th></th><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11 4</th><th>3 2 1 0</th></tr>
<tr><td>0-3 (16entry)</td><td colspan="18">Ignored</td><td colspan="2">Index</td><td>Reserved</td><td>4’h0,1,2,3</td></tr>
<tr><td>0-3 (32entry)</td><td colspan="17">Ignored</td><td colspan="3">Index</td><td>Reserved</td><td>4’h0,1,2,3</td></tr>
<tr><td>4 (1MB)</td><td colspan="10">Ignored</td><td colspan="2">Index</td><td colspan="8">Ignored</td><td>Reserved</td><td>4’h4</td></tr>
<tr><td>4 (4MB)</td><td colspan="8">Ignored</td><td colspan="2">Index</td><td colspan="10">Ignored</td><td>Reserved</td><td>4’h4</td></tr>
<tr><td>4 (16MB)</td><td colspan="6">Ignored</td><td colspan="2">Index</td><td colspan="12">Ignored</td><td>Reserved</td><td>4’h4</td></tr>
<tr><td>4 (64MB)</td><td colspan="4">Ignored</td><td colspan="2">Index</td><td colspan="14">Ignored</td><td>Reserved</td><td>4’h4</td></tr>
<tr><td>5 (Fixed)</td><td colspan="4">Ignored</td><td>Ix</td><td colspan="15">Ignored</td><td>Reserved</td><td>4’h5</td></tr>
<tr><td>5 (128MB)</td><td colspan="3">Ignored</td><td colspan="2">Index</td><td colspan="15">Ignored</td><td>Reserved</td><td>4’h5</td></tr>
<tr><td>5 (256MB)</td><td colspan="2">Ig</td><td colspan="2">Index</td><td colspan="16">Ignored</td><td>Reserved</td><td>4’h5</td></tr>
<tr><td>6 (Fixed)</td><td colspan="3">Ignored</td><td>Ix</td><td colspan="16">Ignored</td><td>Reserved</td><td>4’h6</td></tr>
<tr><td>6 (512MB)</td><td colspan="3">Index</td><td colspan="17">Ignored</td><td>Reserved</td><td>4’h6</td></tr>
<tr><td>6 (256MB)</td><td>Ig</td><td colspan="3">Index</td><td colspan="16">Ignored</td><td>Reserved</td><td>4’h6</td></tr>
<tr><td>7-9(DTLB)</td><td colspan="20">Ignored</td><td>Reserved</td><td>4’h7,8,9</td></tr>
<tr><td></td><td>31</td><td></td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td></td><td></td><td></td><td>15</td><td>14</td><td>13</td><td>12</td><td>11 4</td><td>3 2 1 0</td></tr>
</table>

Figure 4–35. MMU Option Addressing (`as`) Format for `RxTLB0` and `RxTLB1`

Because reading generates more information than can fit in one 32-bit register, there are two read instructions that return different values. The data resulting from the `RITLB0` and `RDTLB0` instructions is shown in Figure 4–36. The low bits contain the `ASID` stored with the entry, while the upper bits contain the Virtual Page Number (VPN) without the Index bits that were used in the address of the read. Unused bits at Bit 12 and above of the data result of these instructions are defined to be zero so that the entire 20-bit field may always be used as a VPN whatever the size of the way. Unused bits at Bit 11 and below are undefined for forward compatibility.

Way	31 30 29 28 27 26 25 24 23 22 21	15 14 13 12	11 8	7 0
0-3 (16entry)	VPN without Index	2'b00	Undefined	ASID
0-3 (32entry)	VPN withoutIndex	3'b000	Undefined	ASID
4 (1MB)	VPN without Index	10'h000	Undefined	ASID
4 (4MB)	VPN without Index	12'h000	Undefined	ASID
4 (16MB)	VPN without Index	14'h0000	Undefined	ASID
4 (64MB)	VPN w/o Idx	16'h0000	Undefined	ASID
5 (Fixed)	4'b1101	16'h0000	Undefined	ASID
5 (128MB)	VPN	17'h00000	Undefined	ASID
5 (256MB)	VPN	18'h00000	Undefined	ASID
6 (Fixed)	3'b111	17'h00000	Undefined	ASID
6 (512MB)	20'h00000		Undefined	ASID
6 (256MB)	V	19'h00000	Undefined	ASID
7-9(DTLB)	VPN		Undefined	ASID
	31 30 29 28 27 26 25 24 23 22 21	15 14 13 12	11 8	7 0

Figure 4–36. MMU Option Data (`at`) Format for `RxTLB0`

The data resulting from the `RITLB1`, and `RDTLB1` instructions is shown in Figure 4–37. The low order four bits contain the attribute stored with the TLB entry (Section 4.6.5.10). The upper bits contain the Physical Page Number (PPN) of the entry. Unused bits at Bit 12 and above of the data result of these instructions are defined to be zero so that the entire 20-bit field may always be used as a PPN, whatever the size of the way. Unused bits at Bit 11 and below are undefined for forward compatibility.

Way	31 29 28 27 26 25 24 23 22 21 20 19 12		11 4	3 0
0-3 (16entry)	PPN		Undefined	Attribute
0-3 (32entry)	PPN		Undefined	Attribute
4 (1MB)	PPN	8’h00	Undefined	Attribute
4 (4MB)	PPN	10’h000	Undefined	Attribute
4 (16MB)	PPN	12’h000	Undefined	Attribute
4 (64MB)	PPN	14’h0000	Undefined	Attribute
5 (Fixed)	5’b00000	15’h0000	Undefined	Attribute
5 (128MB)	PPN	15’h0000	Undefined	Attribute
5 (256MB)	PPN	16’h0000	Undefined	Attribute
6 (Fixed)	4’b1111	16’h0000	Undefined	Attribute
6 (512MB)	PPN	17’h0000	Undefined	Attribute
6 (256MB)	PPN	16’h0000	Undefined	Attribute
7-9(DTLB)	PPN		Undefined	Attribute
	31 29 28 27 26 25 24 23 22 21 20 19 12		11 4	3 0

Figure 4–37. MMU Option Data (`at`) Format for `RxTLB1`

4.6.5.7 Formats for Probing MMU Option TLB Entries

Probing the TLB with the `PITLB` and `PDTLB` instructions requires the formats for the `as` and `at` registers shown in Figure 4–38 and Figure 4–39. Unlike writing and reading the TLBs as explained in the previous two sections, the operation of probing a TLB begins without knowing the way containing the sought after value. The formats do not, therefore, vary with the way being accessed. The probe instructions answer the question of what entry in this TLB, if any, would be used to translate an access with a particular address from a particular ring. The sought for address is given in the `as` register as shown in Figure 4–38 and the ring is given by `PS.RING` (not `CRING`, so that while `PS.EXCM` is set, a probe may be done for a user program). If, for example, there is an entry that matches in address, but its `ASID` does not match any `ASID` in the `RASID` register, or an entry that matches in address, but the `ASID` corresponds in the `RASID` register to a ring of lower number than the current `PS.RING`, the probe will not return a hit.

The format of the `as` register used for the `PITLB` and `PDTLB` instructions is shown in Figure 4–38. Any address may be used as input to the probe instructions.

31 0
Probe Address
32

Figure 4–38. MMU Option Addressing (`as`) Format for `PxTLB`

The data resulting from the `PITLB` and `PDTLB` instructions is shown in Figure 4–39 and Figure 4–40. The low three/four bits contain the Way (if any), which would be used to translate the address and the next bit up is set if there is a translation in the TLB, and clear if there is not. Some bits are undefined for forward compatibility but the result is such that, if Hit=1, it may be used as the `as` register for `WxTLB`, `RxTLB0`, `RxTLB1`, or `IxTLB`.

31 12	11 4	3	2 0
VPN	Undefined	Hit	Way
20	8	1	3

Figure 4–39. MMU Option Data (`at`) Format for `PITLB`

31 12	11 5	4	3 0
VPN	Undefined	Hit	Way
20	7	1	4

Figure 4–40. MMU Option Data (`at`) Format for `PDTLB`

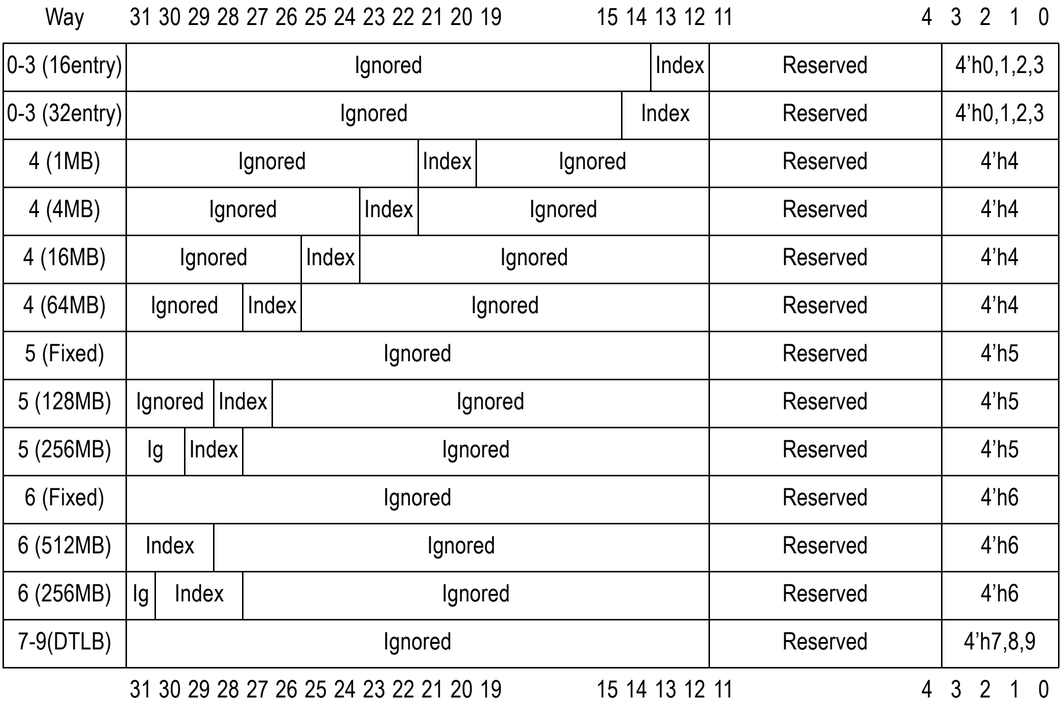
4.6.5.8 Format for Invalidating MMU Option TLB Entries

Invalidating the TLB with the `IITLB` and `IDTLB` instructions requires the formats for the `as` register shown in Figure 4–41. This figure shows, in parallel, the formats for different ways of the cache and different conditions. For Ways 0-3, there are two conditions that depend on the configuration parameter `NIREFILLENTRIES` or `NDREFILLENTRIES` (Figure 4–105) and can have the values of 16 or 32 auto-refill entries per TLB (4 or 8 per TLB way). For Way 4, there are four conditions, which are the four values of the respective`ITLBCFG` or `DTLBCFG` fields and indicate the size of pages currently contained within that way. Ways 5 and 6 can be Fixed or Variable as determined by the `IVARWAY56` and `DVARWAY56` parameters. If they are variable then there are still two conditions which are the two values of the respective `ITLBCFG` or `DTLBCFG` fields and indicate the size of pages currently contained within that way. Each row, then, contains the format for the way and condition indicated in the left column. Note that invalidating Way-5 and Way-6 when the IVARWAY56 and DVARWAY56 parameters are "Fixed" causes no changes because those ways are constant.

Invalidation of ITLB ways 7-15 or DTLB ways 10-15 is undefined.

The format of the as register used for the IITLB and IDTLB instructions is shown in Figure 4–41. The low order four bits contain the way to be accessed. The upper bits contain at least the Index from the Virtual Page Number (VPN). Note that unused bits at Bit 12 and above are ignored so that those bits may simply contain the address for access to all ways of both TLBs. Unused bits at Bit 11 and below are reserved for forward compatibility. They may either be zero or they may be the result of the probe instruction (Section 4.6.5.7 on page 171).

Invalidation of an entry sets the corresponding ASID to zero so that it no longer responds when an address is looked up in the TLB.



Way	31 30 29 28 27 26 25 24 23 22 21 20 19 15 14 13 12			11 … 4	3 2 1 0
0-3 (16entry)	Ignored	Index		Reserved	4’h0,1,2,3
0-3 (32entry)	Ignored	Index		Reserved	4’h0,1,2,3
4 (1MB)	Ignored	Index	Ignored	Reserved	4’h4
4 (4MB)	Ignored	Index	Ignored	Reserved	4’h4
4 (16MB)	Ignored	Index	Ignored	Reserved	4’h4
4 (64MB)	Ignored	Index	Ignored	Reserved	4’h4
5 (Fixed)	Ignored			Reserved	4’h5
5 (128MB)	Ignored	Index	Ignored	Reserved	4’h5
5 (256MB)	Ig	Index	Ignored	Reserved	4’h5
6 (Fixed)	Ignored			Reserved	4’h6
6 (512MB)	Index	Ignored		Reserved	4’h6
6 (256MB)	Ig	Index	Ignored	Reserved	4’h6
7-9(DTLB)	Ignored			Reserved	4’h7,8,9
	31 30 29 28 27 26 25 24 23 22 21 20 19 15 14 13 12			11 … 4	3 2 1 0

Figure 4–41. MMU Option Addressing (as) Format for IxTLB

After modifying any TLB entry with a IITLB instruction, an ISYNC must be executed before executing any instruction that depends on the modification. After modifying any TLB entries with an IDTLB instruction, the operation of loads from and stores that depend on that TLB entry are undefined until a DSYNC instruction is executed.

4.6.5.9 MMU Option Auto-Refill TLB Ways and PTE Format

When no TLB entry matches the ASIDs and the virtual address presented to the MMU, the MMU attempts to automatically load the appropriate page table entry (PTE) from the page table and write it into the TLB in one of the AutoRefill ways. This hardware- generated load from the page table itself requires virtual-to-physical address translation, which executes at Ring 0 so that it has access to the page table and uses the DTLB. An error of any sort during the automatic refill process will cause an InstTLBMissCause or a LoadStoreTLBMissCause exception to be raised so that system software can take appropriate action and possibly retry the access. This combination of hardware and software refill gives excellent performance while minimizing processor complexity. If the second translation succeeds, the PTE load is done through the DataCache, if one is configured, and the attributes for the page containing the PTE enable such a cache access. The PTE's Ring field is then used as an index into the RASID register, and the resulting ASID is written together with the rest of the PTE into the TLB.

Xtensa's TLB refill mechanism requires the page table for the current address space to reside in the current virtual address space. The PTEBase field of the PTEVADDR register gives the base address of the page table. On a TLB miss, the processor forms the virtual address of the PTE by catenating the PTEBase portion of PTEVADDR, the Virtual Page Number (VPN) bits of the miss virtual address, and 2 zero bits. The bits used from PTEVADDR and from the virtual address are configuration dependent; the exact calculation for 4-byte PTEs is

$$\mathtt{PTEVADDR}_{31..22} \| \mathtt{vAddr}_{31..12} \| \mathtt{2'b00}$$

The format of the PTEs is shown in Figure 4–42. The most significant bits hold the Physical Page Number (PPN), the translation of the virtual address corresponding to this entry. The Sw bits are available for software use in the page table (they are not stored in the TLB). The Ring field specifies the privilege level required to access this page; this is used to choose one of the four ASIDs from RASID when the TLB is written. The attribute field gives the access attributes for this page (see Section 4.6.5.10).

31 12	11 6	5 4	3 0
PPN	Sw	Ring	Attribute
20	6	2	4

Figure 4–42. MMU Option Page Table Entry (PTE) Format

The configuration described in Section 4.6.5.4 (with IVARWAY56/DVARWAY56 Fixed) provides a maximum of 3328 MB of dynamically mapped space (4 GB of total virtual address space with 768 MB of statically mapped space). The page table for this maximum size requires 851968 PTEs (3328MB/4 kB). The entire set of PTEs require 3328 kB of virtual address space (at 4 bytes per PTE). The PTEs themselves are at virtual addresses and, therefore, 832 of the PTEs in the table are for mapping the page table itself. These PTEs for mapping the page table will fit onto a single page, the mapping for which may be written into one of the single-entry ways (Ways 7-9) of the data TLB for guaranteed access.

For example, if PTEVADDR is set to 32′hCFC00000, then the virtual address space between there and 32′hCFF3FFFF is used as the page table. That page table is mapped by the 832 entries between 32′hCFF3F000 and 32′hCFF3FCFF. The translation for the page at 32′hCFF3F000 is placed in one of the single-entry ways of the data TLB. (The accesses that might have used the remaining 192 PTE entries on that page would already have been translated by one of the constant ways.) Many of those 832 entries may be marked invalid and the physical address space required for the page table may be made very small.

In systems with large memories, the above maximum configuration may be improved in performance by mapping the entire page table into the constant way (Way 5). If PTEVADDR is set to 32′hD4000000, for example, the virtual address space between there and 32′hD433FFFF, which maps to the physical address space between 32′h04000000 and 32′h0433FFFF (between 64 MB and about 68 MB) is used for a flat page table mapping all of memory. Any TLB miss will now be handled by the hardware refill as the translation for the PTE will be handled by the constant way. The disadvantage is that over 3 MB of memory must be allocated to the page table.

In a small system, where all processes are limited to the first 8 MB of virtual space, PTEVADDR might be set to 32′hCFC00000 and two of the single entry ways set to map the page at 32′hCFC00000 and the page at 32′hCFC01000. One or both pages of PTEs could be used for translations and the hardware refill would always succeed for legal addresses.

4.6.5.10 MMU Option Memory Attributes

Currently available hardware supports the memory attributes described in this section. T1050 hardware supported somewhat different memory attributes, which are described in Section A.5 “MMU Option Memory Attributes”. System software may use the subset of attributes (1, 3, 5, 7, 12, 13, and 14) which have not changed to support all Xtensa processors.

The memory attributes discussed in this section apply both to attribute values written in and read from the TLBs (see Section 4.6.5.5 and Section 4.6.5.6) and to attribute values stored in the PTE entries and written into the AutoRefill ways of the TLBs (see Section 4.6.5.9).

For a more detailed description of the memory access process and the place of these attributes in it, see Section 4.6.2.

Table 4–109 shows the meanings of the attributes for instruction fetch, data load, and data store. For a more detailed description of the memory access process and the place of these attributes in it, see Section 4.6.2.

The first column in Table 4–109 indicates the attribute from the TLB while the remaining columns indicate various effects on the access. The columns are described in the following bullets:

- **Attr** — the value of the 4-bit Attribute field of the TLB entry.
- **Rights** — whether the TLB entry may successfully translate a data load, a data store, or an instruction fetch.
    - The first character is an `r` if the entry is valid for a data load and a dash ("`-`") if not.
    - The second character is a `w` if the entry is valid for a data store and a dash ("`-`") if not.
    - The third character is an `x` if the entry is valid for an instruction fetch and a dash ("`-`") if not.

  If the translation is not successful, an exception is raised.

  Local memory accesses (including XLMI) consult only the Rights column.
- **WB** — some rows are split by whether or not the configured cache is writeback or not. Rows without an entry apply to both cache types.
- **Meaning for Cache Access** — the verbal description of the type of access made to the cache.
- **Access Cache** — indicates whether the cache provides the data.
    - The first character is an `h` if the cache provides the data when the tag indicates hit and a dash ("`-`") if it does not.
    - The second character is an `m` if the cache provides the data when the tag indicates a miss and a dash ("`-`") if it does not. This capability is used only for Isolate mode.

- **Fill Cache** — indicates whether an allocate and fill is done to the cache if the tag indicates a miss.
  - The first character is an `r` if the cache is filled on a data load and a dash ("-") if it is not.
  - The second character is a `w` if the cache is filled on a data store and a dash ("-") if it is not.
  - The third character is an `x` if the cache is filled on an instruction fetch and a dash ("-") if it is not.
- **Guard Load** — refers to the guarded attribute as described in Table 4–99 on page 144. Stores are always guarded and instruction fetches are never guarded, but loads are guarded where there is a “yes” in this column. Local memory loads are not guarded.
- **Write Thru** — indicates whether a write is done through the PIF interface.
  - The first character is an `h` if a Write Thru occurs when the tag indicates hit and a dash ("-") if it does not.
  - The second character is an `m` if a Write Thru occurs when the tag indicates a miss and a dash ("-") if it does not.

Writes to local memories are never Write-Thru. In most implementations, a write-thru will only occur after any needed cache fill is complete.

Table 4–109. MMU Option Attribute Field Values

Attr	Rights	Meaning for Cache Access	Access Cache	Fill Cache	Guard Load	Write Thru
0	r--	Bypass cache	--	---	yes	--
1	r-x	Bypass cache	--	---	yes	--
2	rw-	Bypass cache	--	---	yes	hm
3	rwx	Bypass cache	--	---	yes	hm
4	r--	Cached, WrtBack alloc	h-	r--	-	--
5	r-x	Cached, WrtBack alloc	h-	r-x	-	--
6	rw-	Cached, WrtBack alloc	h-	rw-	-	--
7	rwx	Cached, WrtBack alloc	h-	rwx	-	--
8	r--	Cached, WrtThru	h-	r--	-	--
9	r-x	Cached, WrtThru	h-	r-x	-	--
10	rw-	Cached, WrtThru	h-	r--	-	hm
11	rwx	Cached, WrtThru	h-	r-x	-	hm
12	---	illegal[1]	--	---	-	--
13	rw-	Cache Isolated[2]	hm	---	-	--
14	---	illegal[1]	--	---	-	--
15	---	Reserved[1]	—	—	—	—

[1] Raises exception. EXCCAUSE is set to InstFetchProhibitedCause, LoadProhibitedCause, or StoreProhibitedCause depending on access type

[2] For test only, implementation dependent, uses data cache like local memories and ignores tag.

In the absence of the Instruction Cache Option, Cached regions behave as Bypass regions on instruction fetch. In the absence of the Data Cache Option, Cached regions behave as Bypass regions on data load or store. If the Data Cache is not configured as writeback (Section 4.5.5.1 on page 119) Attributes 4, 5, 6, and 7 behave as Attributes 8, 9, 10, and 11 respectively instead of as they are listed in Table 4–109.

4.6.5.11 MMU Option Operation Semantics

The following functions are used in the operation sections of the individual instruction definitions:

```
function ltranslate(vAddr, ring)
     ltranslate ← (pAddr, attributes, cause)
endfunction ltranslate

function ASID(ring)
     ASID ← $\text{RASID}_{ring*8+ASIDBits-1..ring*8}$
endfunction ASID
```

```
function InstPageBits(wi)
     sizecodebits ← ceil(log2(InstTLB[wi].PageSizeCount))
     sizecode ← $\mathrm{IPAGESIZE}_{wi*4+sizecodebits-1..wi*4}$
     InstPageBits ← InstTLB[wi].PageBits[sizecode]
endfunction InstPageBits

function SplitInstTLBEntrySpec(spec)
     wih ← ceil(log2(InstTLBWayCount)) − 1
     wi ← $\mathrm{spec}_{wih..0}$
     eil ← InstPageBits(wi)
     eih ← eil + log2(InstTLB[wi].IndexCount)
     ei ← $\mathrm{spec}_{eih..eil}$
     vpn ← $\mathrm{spec}_{\mathrm{InstTLBVAddrBits}-1..eih+1}$
     SplitInstTLBEntrySpec ← (vpn, ei, wi)
endfunction SplitInstTLBEntrySpec

function ProbeInstTLB (vAddr)
     match ← 0
     vpn ← undefined
     ei ← undefined
     wi ← undefined
     for i in 0..InstTLBWayCount-1 do
          if then
               match ← match + 1
               vpn ← x
               ei ← x
               wi ← i
          endif
     endfor
     ProbeInstTLB ← (match, vpn, ei, wi)
endfunction ProbeInstTLB
```

4.7 Options for Other Purposes

This section contains options that do not fit easily into the previous sections. The Windowed Register Option provides the hardware for a memory efficient ABI. The Processor Interface Option provides a standard interface to system memory. The Miscellaneous Special Registers Option provides additional scratch registers. The Processor ID Option provides the ability for software to determine on which processor it is running. The Debug Option provides hardware to assist in debugging processors.

4.7.1 Windowed Register Option

The Windowed Register Option replaces the simple 16-entry AR register file with a larger register file from which a window of 16 entries is visible at any given time. The window is rotated on subroutine entry and exit, automatically saving and restoring some registers. When the window is rotated far enough to require registers to be saved to or restored from the program stack, an exception is raised to move some of the register values between the register file and the program stack. The option reduces code size and increases performance of programs by eliminating register saves and restores at procedure entry and exit, and by reducing argument-shuffling at calls. It allows more local variables to live permanently in registers, reducing the need for stack-frame maintenance in non-leaf routines.

Xtensa ISA register windows are different from register windows in other instruction sets. Xtensa register increments are 4, 8, and 12 on a per-call basis, not a fixed increment as in other instruction sets. Also, Xtensa processors have no global address registers. The caller specifies the increment amount, while the callee performs the actual increment by the `ENTRY` instruction. The compiler uses an increment sufficient to hide the registers that are live at the point of the call (which the compiler can pack into the fewest possible at the low end of the register-number space). The number of physical registers is 32 or 64, which makes this a more economical configuration. Sixteen registers are visible at one time. Assuming that the average number of live registers at the point of call is 6.5 (return address, stack pointer, and 4.5 local variables), and that the last routine uses 12 registers at its peak, this allows nine call levels to live in 64 registers ($8\times6.5+12=64$). As an example, an average of 6.5 live registers might represent 50% of the calls using an increment of 4, 38% using an increment of 8, and 12% using an increment of 12.

- Prerequisites: Exception Option (page 82)
- Incompatible options: None

The rotation of the 16-entry visible window within the larger register file is controlled by the WindowBase Special Register added by the option. The rotation always occurs in units of four registers, causing the number of bits in WindowBase to be $\log_2$(NAREG/4). Rotation at the time of a call can instantly save some registers and provide new registers for the called routine. Each saved register has a reserved location on the stack, to which it may be saved if the call stack extends enough farther to need to re-use the physical registers. The WindowStart Special Register, which is also added by the option and consists of NAREG/4 bits, indicates which four register units are currently cached in the physical register file instead of residing in their stack locations. An attempt to use registers live with values from a parent routine raises an Overflow Exception which saves those values and frees the registers for use. A return to a calling routine whose registers have been previously saved to the stack raises an Underflow Exception which restores those values. Programs without wide swings in the depth of the call stack save and restore values only occasionally.

4.7.1.1 Windowed Register Option Architectural Additions

Table 4–110 through Table 4–113 show this option's architectural additions.

Table 4–110. Windowed Register Option Constant Additions (Exception Causes)

Exception Cause	Description	Constant Value
`AllocaCause`	`MOVSP` instruction, if the caller's registers are not present in the register file (seeTable 4–64 on page 89)	`6'b000101` (decimal 5)

Table 4–111. Windowed Register Option Processor-Configuration Additions

Parameter	Description	Valid Values
`WindowOverflow4`	Window overflow exception vector for 4-register stack frame	32-bit address[1]
`WindowUnderflow4`	Window underflow exception vector for 4-register stack frame	32-bit address[1]
`WindowOverflow8`	Window overflow exception vector for 8-register stack frame	32-bit address[1]
`WindowUnderflow8`	Window underflow exception vector for 8-register stack frame	32-bit address[1]
`WindowOverflow12`	Window overflow exception vector for 12- register stack frame	32-bit address[1]
`WindowUnderflow12`	Window underflow exception vector for 12-register stack frame	32-bit address[1]
`NAREG`	Number of address registers	32 or 64

[1] Some implementations have restrictions on the alignment and relative location of the WindowOverflowN and WindowUnderflowN vectors. See "`procedure WindowCheck (wr, ws, wt)`" in Section 4.7.1.3 "Window Overflow Check" on page 184 for how these are used.

Table 4–112. Windowed Register Option Processor-State Additions and Changes

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
`AR`	`NAREG`	32	Address registers (general registers)	R/W	—
`WindowBase`	1	log2(NAREG/4)	Base of current address-register window	R/W	72
`WindowStart`	1	NAREG/4	Call-window start bits	R/W	73

[1] Registers with a Special Register assignment are read and/or written with the `RSR`, `WSR`, and `XSR` instructions. See Table 5–127 on page 205.

Table 4–112. Windowed Register Option Processor-State Additions and Changes

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
PS.CALLINC	1	2	Miscellaneous processor state, window increment from call (see Table 4–63 on page 87)	R/W	230
PS.OWB	1	4	Miscellaneous processor state, old window base (see Table 4–63 on page 87)	R/W	230
PS.WOE	1	1	Miscellaneous processor state, window overflow enable (see Table 4–63 on page 87)	R/W	230

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 5–127 on page 205.

Table 4–113. Windowed Register Option Instruction Additions

Instruction[1]	Format	Definition
MOVSP	RRR	Atomic check window and move
CALL4, CALL8, CALL12	CALL	Call subroutine, PC-relative. These instructions communicate the number of registers to hide using PS.CALLINC in addition to the operation of CALL0.
CALLX4, CALLX8, CALLX12	CALLX	Call subroutine, address in register. These instructions communicate the number of registers to hide using PS.CALLINC in addition to the operation of CALLX0.
ENTRY	BRI12	Subroutine entry—rotate registers, adjust stack pointer. This instruction should not be used in a routine called by CALL0 or CALLX0.
RETW	CALLX	Subroutine return—unrotate registers, jump to return address. Used to return from a routine called by CALL4, CALL8, CALL12, CALLX4, CALLX8, or CALLX12.
RETW.N[2]	RRRN	Same at RETW in a 16-bit encoding
ROTW	RRR	Rotate window by a constant. ROTW is intended for use in exception handlers and context switch.
L32E	RRI4	Load 32 bits for window exception
S32E	RRI4	Store 32 bits for window exception
RFWO	RRR	Return from window overflow exception
RFWU	RRR	Return from window underflow exception

[1] These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.

[2] Exists only if the Code Density Option described in Section 4.3.1 on page 53 is configured.

4.7.1.2 Managing Physical Registers

The WindowBase Special Register gives the position of the current window into the physical register file. In the instruction descriptions, AR[i] is a short-hand for a reference to the physical register file AddressRegister defined as follows:

$$\mathtt{AddressRegister[((2'b00}\|\mathtt{i}_{3..2}\mathtt{)\ +\ WindowBase)}\ \|\ \mathtt{i}_{1..0}\mathtt{]}$$

The WindowStart Special Register gives the state of physical registers (unused or part of a window). WindowStart is used both to detect overflow and underflow on register use and procedure return, as well as to determine the number of registers to be saved in a given stack frame when handling exceptions and switching contexts. There is one bit in WindowStart for each four physical registers. This bit is set if those four registers are AR[0] to AR[3] for some call. WindowStart bits are set by ENTRY and cleared by RETW.

The WindowBase and WindowStart registers are undefined after processor reset, and should be initialized by the reset exception vector code.

Figure 4–43 through Figure 4–45 show three functionally identical implementations of windowed registers. Figure 4–43 shows the concept of how the registers are addressed. Figure 4–44 shows logic with the same functional result but with little or no penalty paid in timing for the addition of the WindowBase value. Figure 4–45 shows a third version of the logic with the same functional result but with no timing loss at all caused by the addition of the WindowBase value.

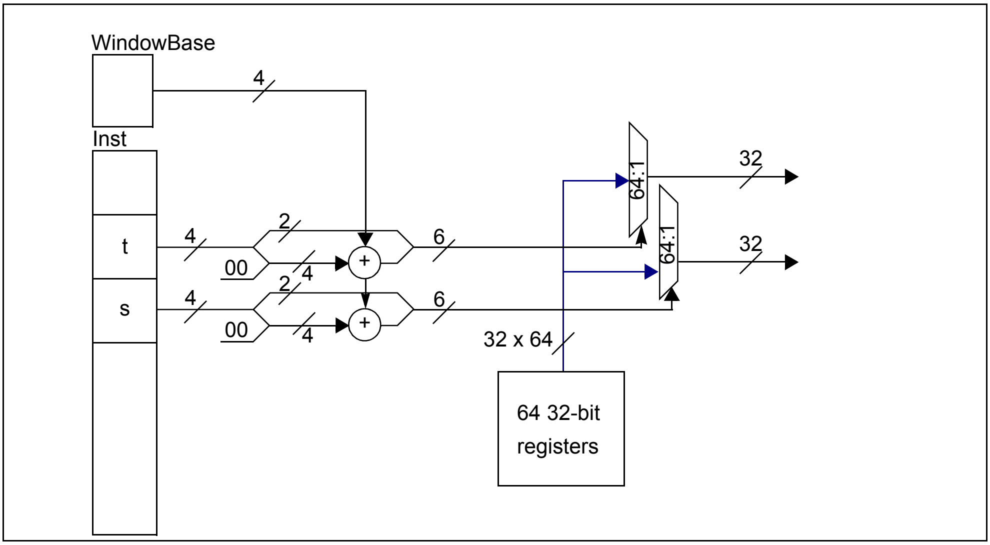


Figure 4–43. Conceptual Register Window Read

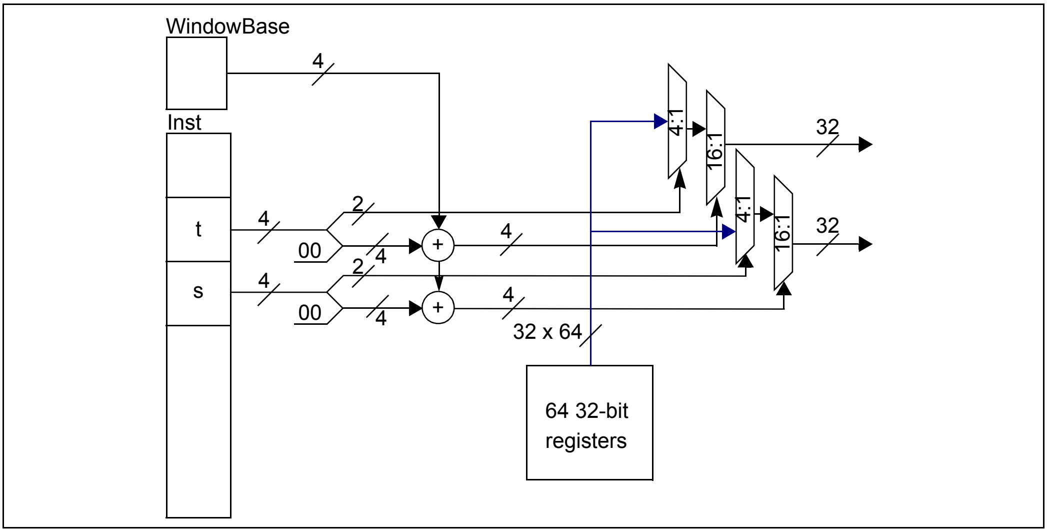


Figure 4–44. Faster Register Window Read

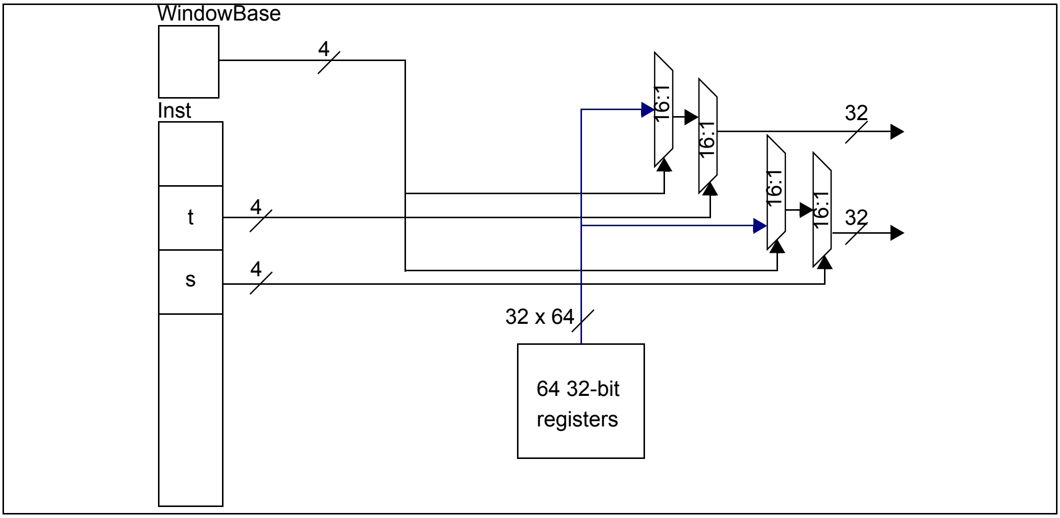


Figure 4–45. Fastest Register Window Read

4.7.1.3 Window Overflow Check

The `ENTRY` instruction moves the register window, but does not guarantee that all the registers in the current window are available for use. Instead, the processor waits for the first reference to an occupied physical register before triggering a window overflow. This prevents unnecessary overflows, because many routines do not use all 16 of their virtual registers. Figure 4–46 shows the state of the register file just prior to a reference that causes an overflow. The WS(n) notation shows which `WindowStart` bits are set in this example, and gives the distance to the next bit set (that is, the number of registers stored for the corresponding stack frame). In the figure, “rmax” indicates the maximum register that the current procedure uses and “Base” is an abbreviation for `WindowBase`. Note that registers are considered in groups of four here.

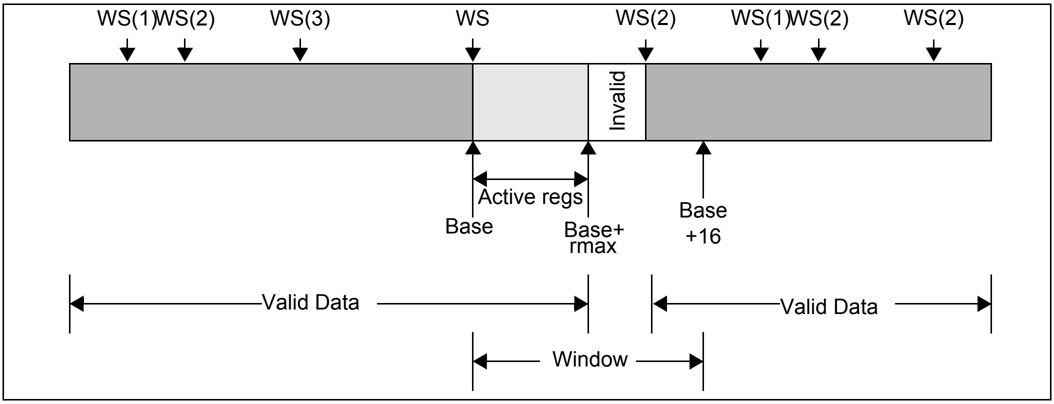


Figure 4–46. Register Window Near Overflow

The check for overflow is done as follows:

```
WindowCheck (  if ref(AR[r]) then $r_{3..2}$ else 2'b00,
               if ref(AR[s]) then $s_{3..2}$ else 2'b00,
               if ref(AR[t]) then $t_{3..2}$ else 2'b00)
```

where `ref()` is 1 if the register is used by the instruction, and 0 otherwise, and WindowCheck is defined as follows:

```
procedure WindowCheck (wr, ws, wt)
     n ← if (wr ≠ 2'b00 or ws ≠ 2'b00 or wt ≠ 2'b00)
               and $WindowStart_{WindowBase+1}$ then 2’b01
          else if ($wr_1$ or $ws_1$ or $wt_1$)
               and $WindowStart_{WindowBase+2}$ then 2’b10
          else if (wr = 2'b11 or ws = 2'b11 or wt = 2'b11)
               and $WindowStart_{WindowBase+3}$ then 2’b11
          else 2’b00
     if CWOE = 1 and n ≠ 2’b00 then
          PS.OWB ← WindowBase
          m ← WindowBase + (2'b00||n)
          PS.EXCM ← 1
          EPC[1] ← PC
          nextPC ← if $WindowStart_{m+1}$ then WindowOverflow4
               else if $WindowStart_{m+2}$ then WindowOverflow8
               else WindowOverflow12
```

```
            WindowBase ← m
        endif
    endprocedure WindowCheck
```

A single instruction may raise multiple window overflow exceptions. For example, suppose that registers 4..7 of the current window still contain a previous call frame's values (WindowStart$_{\text{WindowBase+1}}$ is set), and 8..15 are part of the subroutine called by that frame (WindowStart$_{\text{WindowBase+2}}$ is also set), and an instruction references register 10. The processor will raise an exception to spill registers 4..7 and then return to retry the instruction, which will then raise another exception to spill registers 8..15. On return from this overflow handler, the reference will finally succeed.

4.7.1.4 Call, Entry, and Return Mechanism

The register window mechanics of the {CALL, CALLX}{4,8,12}, ENTRY, and {RETW, RETW.N} instructions are:

```
CALLn/CALLXn
    WindowCheck (2'b00, 2'b00, n)
    PS.CALLINC ← n
    AR[n||2'b00] ← n || (PC + 3)$_{29..0}$

ENTRY s, imm12
    AR[PS.CALLINC||s$_{1..0}$] ← AR[s] − ($0^{17}$||imm12||$0^{3}$)
    WindowBase ← WindowBase + ($0^{2}$||PS.CALLINC)
    WindowStart$_{WindowBase}$ ← 1
```

In the definition of ENTRY above, the AR read and the AR write refer to different registers.

```
RETW/RETW.N
    n ← AR[0]$_{31..30}$
    nextPC ← PC$_{31..30}$ || AR[0]$_{29..0}$
    owb ← WindowBase
    m ← if WindowStart$_{WindowBase-4'b0001}$ then 2'b01
     elsif WindowStart$_{WindowBase-4'b0010}$ then 2'b10
     elsif WindowStart$_{WindowBase-4'b0011}$ then 2'b11
     else 2'b00
    if n = 2'b00 | (m ≠ 2'b00 & m ≠ n) | PS.WOE=0 | PS.EXCM=1 then
        -- undefined operation
        -- may raise illegal instruction exception
    else
        WindowBase ← WindowBase − ($0^{2}$||n)
        if WindowStart$_{WindowBase}$ ≠ 0 then
            WindowStart$_{owb}$ ← 0
        else
            -- Underflow exception
            PS.EXCM ← 1
```

```
                    EPC[1] ← PC
                    PS.OWB ← owb
                    nextPC ← if n = 2'b01 then WindowUnderflow4
                              else if n = 2'b10 then WindowUnderflow8
                              else WindowUnderflow12
                endif
            endif
```

The RETW opcode assignment is such that the s and t fields are both zero, so that the hardware may use either AR[s] or AR[t] in place of AR[0] above. Underflow is detected by the caller's window's WindowStart bit being clear (that is, not valid). Figure 4–47 shows the register file just before a RETW that raises an underflow exception. window overflow and window underflow exceptions leave PS.UM unchanged.

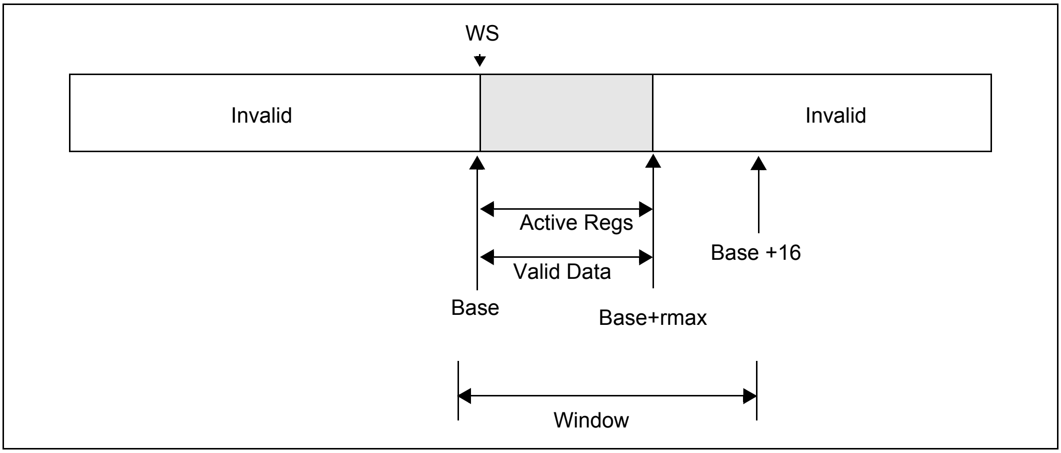


Figure 4–47. Register Window Just Before Underflow

4.7.1.5 Windowed Procedure-Call Protocol

While the procedure-call protocol is a matter for the compiler and ABI, the Xtensa ISA, and particularly the Windowed Register Option was designed with the following goals in mind:

- Provide highly efficient call/return (measured in both code size and execution time)
- Support per-call register window increments
- Use a single stack for both register save/restore and local variables
- Support variable frame sizes (for example, alloca)
- Support programming language exception handling (for example, setjmp/longjmp, catch/throw, and so forth)
- Support debuggers

- Require minimal special ISA features (special registers and so forth)

Table 4–114 shows the register usage in the Windowed Register Option. Refer to Section 8.1 “The Windowed Register and CALL0 ABIs” for a more complete description of the Windowed Register ABI.

Table 4–114. Windowed Register Usage

Callee Register	Register Name	Usage
0	`a0`	Return address
1	`a1/sp`	Stack pointer
2..7	`a2..a7`	In, out, inout, and return values

Calls to routines that use only `a2..a3` as parameters may use the `CALL4`, `CALL8`, or `CALL12` instructions to save 4, 8, or 12 live registers. Calls to routines that use `a2..a7` for parameters may use only `CALL4` or `CALL8`. The following assembly language illustrates the call protocol.

```
// In procedure g, the call
//   z = f(x, y)
// would compile into
     mov       a6, x     // a6 is f’s a2 (x)
     mov       a7, y     // a7 is f’s a3 (y)
     call4     f         // put return address in f’s a0,
                         // goto f
     mov       z, a6     // a6 is f’s a2 (return value)
// The function
//   int f(int a, int *b) { return a + *b; }
// would compile into
f:   entry     sp, framesize// allocate stack frame, rotate regs
     // on entry, a0/ return address, a1/ stack pointer,
     // a2/ a, a3/ *b
     l32i      a3, a3, 0 // *b
     add       a2, a2, a3// *b + a
     retw
```

The “highly efficient call/return” goal requires that there not be separate stack and frame pointer registers in cases where they would differ by a constant (that is, no `alloca` is used). There are simply not enough registers to waste. For routines that do call `alloca`, the compiler will copy the initial stack pointer to another register and use that for addressing all locals.

The variable allocation,

```
p1 = alloca(n1);
```

will be implemented as

```
movi     t4, -16          // for alignment to 16-byte boundary
sub      t5, sp, n1       // reserve stack space
and      t4, t5, t4       // ...
movsp    sp, t4           // atomically set sp
addi     p1, sp, -16+botsize// save pointer
```

The `botsize` in the last statement allows the compiler to maintain a block of words at the bottom of the stack (for example, this block might be for memory arguments to routines). The -16 is a constant of the call protocol; it puts 16 bytes of the bottom area below the stack pointer (since they are infrequently referenced), leaving the limited range of the ISA's load/store offsets available for more frequently referenced locals.

Figure 4–48 and Figure 4–49 show the stack frame before and after `alloca`.

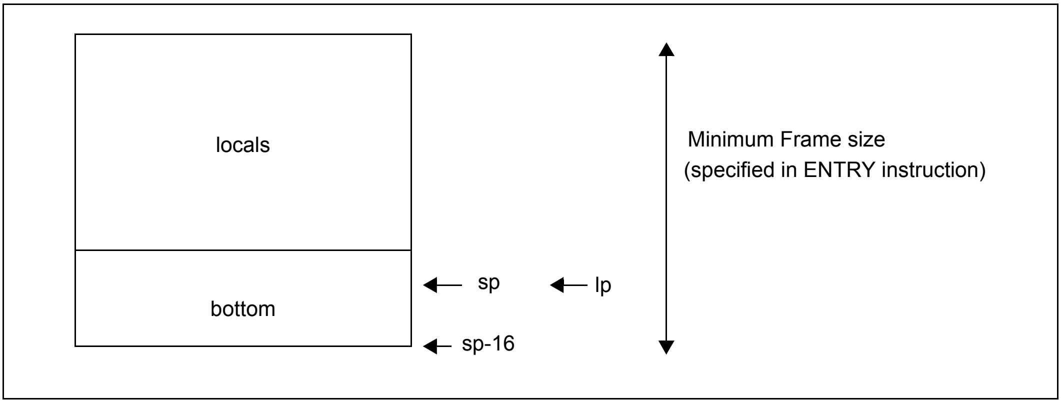


Figure 4–48. Stack Frame Before `alloca()`

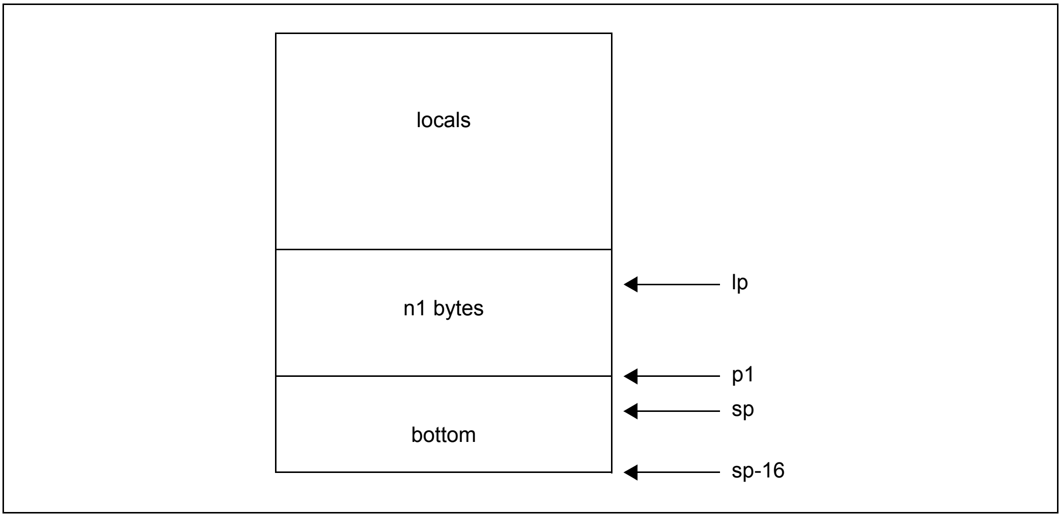


Figure 4–49. Stack Frame After First `alloca()`

Figure 4–50 shows the stacking of frames when the stack grows downward, as on most other systems. The window save area for a frame is addressed with negative offsets from the next stack frame's `sp`. Four registers are saved in the base save area. If more than four registers are saved, they are stored at the top of the stack frame, in the extra save area, which can be found with negative offsets from the previous stack frame's `sp`. This unusual split allows for simple backtrace while providing for a variable sized save area.

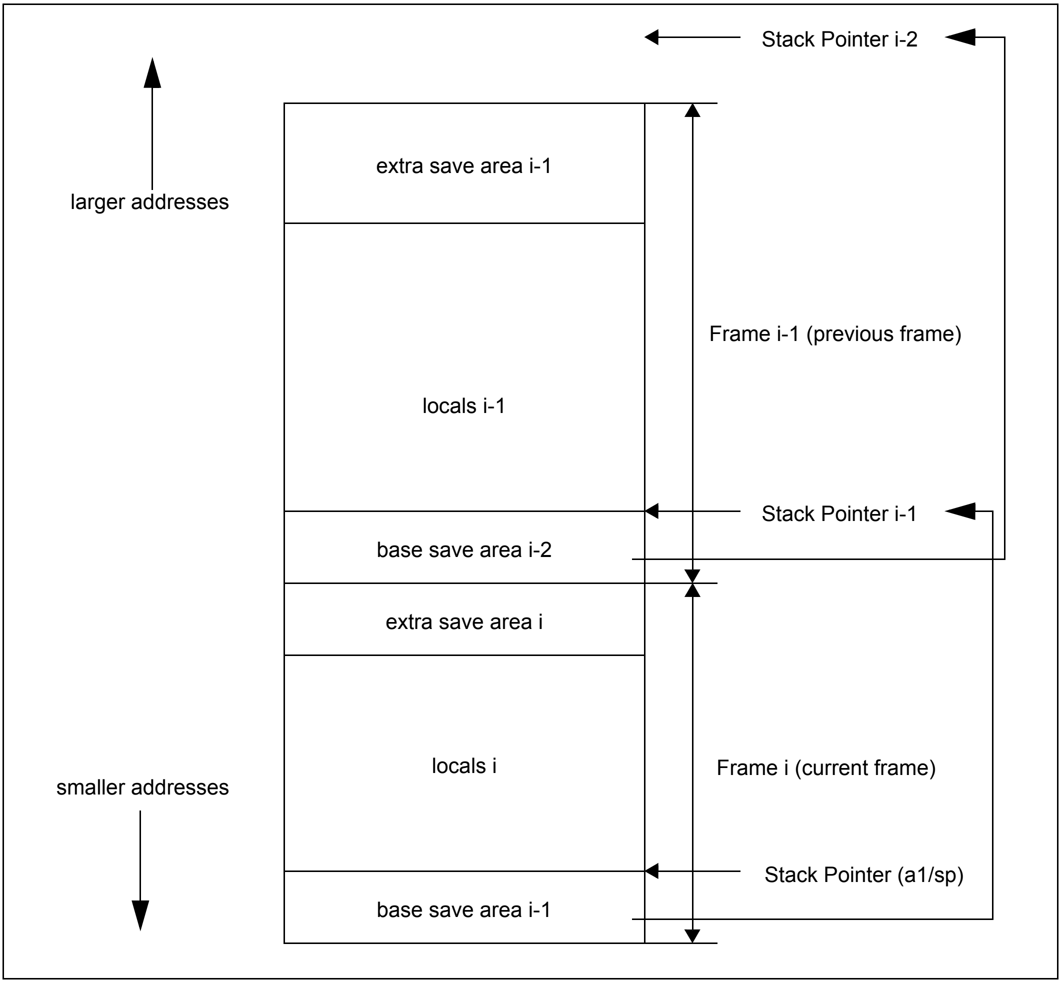


Figure 4–50. Stack Frame Layout

Several of the goals listed on page 187 require that call stacks be backward-traceable. That is, from the state of `call[i]`, it must be possible to determine the state of `call[i-1]`. It is best if the state of `call[i]` can be summarized in a single pointer (at least when the registers have been saved), in which case this requirement is best described as: There must be a means of determining the pointer for `call[i-1]` from the pointer of `call[i]`. For managing register-window overflow or underflow, this method should also be very efficient; it should not, for example, involve routine-specific information or other table lookup (for example, frame size or stack offsets).

The Xtensa ISA represents the state of `call[i]` with its stack pointer (not the frame pointer, as that is routine-specific and would cost too much to lookup). This can be made to work even with `alloca`. Therefore it must be possible to read the stack pointer for call[i-1] at a fixed offset from the stack pointer (not the frame pointer) for `call[i]`. Thus, the stack pointer for `call[i-1]` is stored in the area labeled "base save area i-1" in Figure 4–48.

For efficiency, the `call[i-1]` stack pointer is only stored into `call[i]`'s frame when `call[i-1]`'s registers are stored into the stack on overflow. This is sufficient for register window underflow handling. Other back-tracing operations should begin by storing registers of all call frames back into the stack.

Because the `call[i-1]` stack pointer is referenced infrequently, it is stored at a negative offset from the stack pointer. This leaves the ISA's limited positive offsets available for more frequent uses. Thus, the stack always reaches to 16 bytes below the contents of the stack pointer. Interrupts and such must respect this 16-byte reserved space below the stack pointer. Because the minimum number of registers to save is four, the processor stores four of `call[i-1]`'s registers, `a0..a3`, in this space; the rest (if any) are saved in `call[i-1]`'s own frame.

The register-window call instructions only store the least-significant 30 bits of the return address. Register-window return instructions leave the two most-significant bits of the PC unchanged. Therefore, subroutines called using register window instructions must be placed in the same 1 GB address region as the call.

4.7.1.6 Window Overflow and Underflow to and from the Program Stack

Register-window underflow occurs when a return instruction decrements to a window that has been spilled (indicated by its `WindowStart` bit being cleared). The processor saves the current `PC` in `EPC[1]` and transfers to one of three underflow handlers based on the register window decrement. When the MMU Option is configured, it is necessary for the handlers to access the stack with the same privilege level as the code that raised the exception. Two special instructions, `L32E` and `S32E`, are therefore added by the Windowed Register Option for this purpose. In addition, these instructions use negative offsets in the formation of the virtual address, which saves several instructions in the handlers. The exception handlers could be as simple as the following:

```
WindowOverflow4:    // inside call[i] referencing a register that
                    // contains data from call[j]
     // On entry here: window rotated to call[j] start point; the
     // registers to be saved are a0-a3; a4-a15 must be preserved
     // a5 is call[j+1]'s stack pointer
     s32e  a0, a5, -16   // save a0 to call[j+1]'s frame
     s32e  a1, a5, -12   // save a1 to call[j+1]'s frame
     s32e  a2, a5,  -8   // save a2 to call[j+1]'s frame
     s32e  a3, a5,  -4   // save a3 to call[j+1]'s frame
     rfwo                // rotates back to call[i] position

WindowUnderflow4:   // returning from call[i+1] to call[i] where
```

```
                    // call[i]'s registers must be reloaded
     // On entry here: a0-a3 are to be reloaded with
     // call[i].reg[0..3] but initially contain garbage.
     // a4-a15 are call[i+1].reg[0..11],
     // (in particular, a5 is call[i+1]'s stack pointer)
     // and must be preserved
     l32e  a0, a5, -16   // restore a0 from call[i+1]'s frame
     l32e  a1, a5, -12   // restore a1 from call[i+1]'s frame
     l32e  a2, a5,  -8   // restore a2 from call[i+1]'s frame
     l32e  a3, a5,  -4   // restore a3 from call[i+1]'s frame
     rfwu

WindowOverflow8:
     // On entry here: window rotated to call[j]; the registers to be
     // saved are a0-a7; a8-a15 must be preserved
     // a9 is call[j+1]'s stack pointer
     s32e  a0, a9, -16   // save a0 to call[j+1]'s frame
     l32e  a0, a1, -12   // a0 <- call[j-1]'s sp
     s32e  a1, a9, -12   // save a1 to call[j+1]'s frame
     s32e  a2, a9,  -8   // save a2 to call[j+1]'s frame
     s32e  a3, a9,  -4   // save a3 to call[j+1]'s frame
     s32e  a4, a0, -32   // save a4 to call[j]'s frame
     s32e  a5, a0, -28   // save a5 to call[j]'s frame
     s32e  a6, a0, -24   // save a6 to call[j]'s frame
     s32e  a7, a0, -20   // save a7 to call[j]'s frame
     rfwo                // rotates back to call[i] position

WindowUnderflow8:
     // On entry here: a0-a7 are call[i].reg[0..7] and initially
     // contain garbage, a8-a15 are call[i+1].reg[0..7],
     // (in particular, a9 is call[i+1]'s stack pointer)
     // and must be preserved
     l32e  a0, a9, -16   // restore a0 from call[i+1]'s frame
     l32e  a1, a9, -12   // restore a1 from call[i+1]'s frame
     l32e  a2, a9,  -8   // restore a2 from call[i+1]'s frame
     l32e  a7, a1, -12   // a7 <- call[i-1]'s sp
     l32e  a3, a9,  -4   // restore a3 from call[i+1]'s frame
     l32e  a4, a7, -32   // restore a4 from call[i]'s frame
     l32e  a5, a7, -28   // restore a5 from call[i]'s frame
     l32e  a6, a7, -24   // restore a6 from call[i]'s frame
     l32e  a7, a7, -20   // restore a7 from call[i]'s frame
     rfwu

WindowOverflow12:
     // On entry here: window rotated to call[j]; the registers to be
     // saved are a0-a11; a12-a15 must be preserved
     // a13 is call[j+1]'s stack pointer
     s32e   a0, a13, -16  // save a0 to call[j+1]'s frame
     l32e   a0,  a1, -12  // a0 <- call[j-1]'s sp
```

```
        s32e   a1, a13, -12  // save a1 to call[j+1]'s frame
        s32e   a2, a13,  -8  // save a2 to call[j+1]'s frame
        s32e   a3, a13,  -4  // save a3 to call[j+1]'s frame
        s32e   a4,  a0, -48  // save a4 to end of call[j]'s frame
        s32e   a5,  a0, -44  // save a5 to end of call[j]'s frame
        s32e   a6,  a0, -40  // save a6 to end of call[j]'s frame
        s32e   a7,  a0, -36  // save a7 to end of call[j]'s frame
        s32e   a8,  a0, -32  // save a8 to end of call[j]'s frame
        s32e   a9,  a0, -28  // save a9 to end of call[j]'s frame
        s32e  a10,  a0, -24  // save a10 to end of call[j]'s frame
        s32e  a11,  a0, -20  // save a11 to end of call[j]'s frame
        rfwo                 // rotates back to call[i] position

   WindowUnderflow12:
        // On entry here: a0-a11 are call[i].reg[0..11] and initially
        // contain garbage, a12-a15 are call[i+1].reg[0..3],
        // (in particular, a13 is call[i+1]'s stack pointer)
        // and must be preserved
        l32e   a0,  a13, -16 // restore a0 from call[i+1]'s frame
        l32e   a1,  a13, -12 // restore a1 from call[i+1]'s frame
        l32e   a2,  a13,  -8 // restore a2 from call[i+1]'s frame
        l32e   a11,  a1, -12 // a11 <- call[i-1]'s sp
        l32e   a3,  a13,  -4 // restore a3 from call[i+1]'s frame
        l32e   a4,  a11, -48 // restore  a4 from end of call[i]'s frame
        l32e   a5,  a11, -44 // restore  a5 from end of call[i]'s frame
        l32e   a6,  a11, -40 // restore  a6 from end of call[i]'s frame
        l32e   a7,  a11, -36 // restore  a7 from end of call[i]'s frame
        l32e   a8,  a11, -32 // restore  a8 from end of call[i]'s frame
        l32e   a9,  a11, -28 // restore  a9 from end of call[i]'s frame
        l32e  a10,  a11, -24 // restore a10 from end of call[i]'s frame
        l32e  a11,  a11, -20 // restore a11 from end of call[i]'s frame
        rfwu
```

4.7.2 Processor Interface Option

The Processor Interface Option adds a bus interface used by memory accesses, which are to locations other than local memories (page 123 through page 126). It is used for cache misses for cacheable addresses (page 111 through page 122), as well as for cache bypass memory accesses.

Direct memory access to local memories from outside may also be configured through the bus interface added by the Processor Interface Option. The direct memory access may either be top priority for highest bandwidth or intermediate priority for greatest efficiency.

- Prerequisites: None
- Incompatible options: None

- Historical note: The additions made by this option were once considered part of the Core Architecture and so compatibility with previous hardware might require the use of this option.

Refer to a specific Xtensa processor data book for more detail on the Processor Interface Option.

4.7.2.1 Processor Interface Option Architectural Additions

Table 4–115 shows this option's architectural additions (seeTable 4–64 on page 89 for more). Note that asynchronous load/store errors are delivered via a configuration-dependent interrupt.

Table 4–115. Processor Interface Option Constant Additions (Exception Causes)

Exception Cause	Description	Constant Value
`InstrPIFDataErrorCause`	PIF data error during instruction fetch	`6'b001100` (decimal 12)
`LoadStorePIFDataErrorCause`	Synchronous PIF data error during LoadStore access	`6'b001101` (decimal 13)
`InstrPIFAddrErrorCause`	PIF address error during instruction fetch	`6'b001110` (decimal 14)
`LoadStorePIFAddrErrorCause`	Synchronous PIF address error during LoadStore access	`6'b001111` (decimal 15)

4.7.3 Miscellaneous Special Registers Option

The Miscellaneous Special Registers Option provides zero to four scratch registers within the processor readable and writable by `RSR`, `WSR`, and `XSR`. These registers are privileged. They may be useful for some application-specific exception and interrupt processing tasks in the kernel. The `MISC` registers are undefined after reset.

- Prerequisites: None
- Incompatible options: None

4.7.3.1 Miscellaneous Special Registers Option Architectural Additions

Table 4–116 and Table 4–117 show this option's architectural additions.

Table 4–116. Miscellaneous Special Registers Option Processor-Configuration Additions

Parameter	Description	Valid Values
`NMISC`	Number of miscellaneous 32-bit Special Registers	0..4

Table 4–117. Miscellaneous Special Registers Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
MISC	NMISC	32	Miscellaneous privileged register	R/W	244-247

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 5–127 on page 205.

4.7.4 Thread Pointer Option

The Thread Pointer Option provides an additional register to facilitate implementation of Thread Local Storage by operating systems and tools. The register is readable and writable by RUR and WUR. The register is unprivileged and is undefined after reset.

- Prerequisites: None
- Incompatible options: None

4.7.4.1 Thread Pointer Option Architectural Additions

Table 4–118 shows this option's architectural additions.

Table 4–118. Thread Pointer Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Register Number[1]
THREADPTR	1	32	Thread pointer	R/W	User 231

[1] See Table 5–127 on page 205.

4.7.5 Processor ID Option

In some applications there are multiple Xtensa processors executing from the same instruction memory, and there is a need to distinguish one processor from another. This option allows the system logic to provide each processor an identity by reading the PRID register. The PRID value for each processor is typically in the range 0..NPROCESSORS-1, but this is not required. The PRID register is constant after reset.

- Prerequisites: None
- Incompatible options: None

4.7.5.1 Processor ID Option Architectural Additions

Table 4–119 shows this option's architectural additions.

Table 4–119. Processor ID Option Special Register Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
PRID	1	32[2]	Processor Id	R	235

[1] Registers with a Special Register assignment are read with the RSR instruction. See Table 5–127 on page 205.

[2] Some implementations may support only the low 16 bits of the PRID register.

4.7.6 Debug Option

The Debug Option implements instruction-counting and breakpoint exceptions for debugging by software or external hardware. The option uses an interrupt level previously defined in the High-Priority Interrupt Option. In some implementations, some debug interrupts may not be masked by PS.INTLEVEL (see the *Tensilica On-Chip Debugging Guide*). The Debug Option is useful when configuring a new (not previously debugged) Xtensa processor configuration or for running previously untested software on a processor.

- Prerequisites: High-Priority Interrupt Option (page 106)
- Incompatible options: None

Some of the features listed below are added only when the OCD Option (see the *Tensilica On-Chip Debugging Guide*) is configured in addition to the Debug Option. Those features are included here, under the Debug Option, so that their architectural aspects are documented, but marked as "available only with OCD Option."

4.7.6.1 Debug Option Architectural Additions

Table 4–120 through Table 4–122 show this option's architectural additions.

Table 4–120. Debug Option Processor-Configuration Additions

Parameter	Description	Valid Values
DEBUGLEVEL	Debug interrupt level	2..NLEVEL[1,2]
NIBREAK	Number of instruction breakpoints (break registers)	0..2
NDBREAK	Number of data breakpoints (break registers)	0..2
SZICOUNT	Number of bits in the ICOUNT register	2, 32

[1] NLEVEL is specified in the High-Priority Interrupt Option, Table 4–74 on page 107.

[2] DEBUGLEVEL must be greater than EXCMLEVEL (see Table 4–74 on page 107)

Table 4–121. Debug Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
ICOUNT	1	2,32	Instruction count	R/W	236
ICOUNTLEVEL	1	4	Instruction-count level	R/W	237
IBREAKA	NIBREAK	32	Instruction-break address	R/W	128-129
IBREAKENABLE	1	NIBREAK	Instruction-break enable bits	R/W	96
DBREAKA	NDBREAK	32	Data-break address	R/W	144-145
DBREAKC	NDBREAK	8[2]	Data break control	R/W	160-161
DEBUGCAUSE	1	10	Cause of last debug exception	R	233
DDR[3]	1[3]	32	Debug data register	R/W	104

1. Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 5–127 on page 205.
2. See Figure 4–52 on page 202 for the DBREAKC register format.
3. The DDR register may have separate physical registers for in and out directions in some implementations. The register is only available with the OCD Option, for which the Debug Option is a prerequisite.

Table 4–122. Debug Option Instruction Additions

Instruction[1]	Format	Definition
BREAK	RRR	Breakpoint
BREAK.N[2]	RRRN	Narrow breakpoint

1. These instructions are fully described in Chapter 6, "Instruction Descriptions" on page 243.
2. Exists only if the Code Density Option described in Section 4.3.1 on page 53 is configured.

4.7.6.2 Debug Cause Register

The DEBUGCAUSE register contains a coded value giving the reason(s) that the processor took the debug exception. It is implementation-specific whether all applicable bits are set or whether lower-priority conditions are undetected in the presence of higher-priority conditions.

For the priority of the bits in the DEBUGCAUSE register, see Section 4.4.1.11.

Figure 4–51 below shows the bits in the DEBUGCAUSE register, and Table 4–123 describes more fully the meaning of each bit.

31 12	11 8	7 6	5	4	3	2	1	0
reserved	DBNUM	re-served	DI	BN	BI	DB	IB	IC
	4				6			

Figure 4–51. DEBUGCAUSE Register

Table 4–123. DEBUGCAUSE Fields

Bit	Field	Meaning
0	IC	`ICOUNT` exception
1	IB	`IBREAK` exception
2	DB	`DBREAK` exception
3	BI	`BREAK` instruction
4	BN	`BREAK.N` instruction
5	DI	Debug interrupt[1]
11-8	DBNUM	Which of the `DBREAK` registers matched (added in RA-2004.1 release)

[1] The debug interrupt is only available with the OCD Option.

The `DEBUGCAUSE` register is undefined after processor reset and when `CINTLEVEL` < `DEBUGLEVEL`.

4.7.6.3 Using Breakpoints

`BREAK` and `BREAK.N` are 24-bit and 16-bit instructions that simply raise a `DEBUGLEVEL` exception with `DEBUGCAUSE` bit 3 or 4 set, respectively, when executed. Software can replace an instruction with a breakpoint instruction to transfer control to a debug monitor when execution reaches the replaced instruction.

The `BREAK` and `BREAK.N` instructions cannot be used on ROM code, and so the ISA provides a configurable number of instruction-address breakpoint registers. When the processor is about to complete the execution of the instruction fetched from virtual address `IBREAKA[i]`, and $\texttt{IBREAKENABLE}_i$ is set, it raises an exception instead. It is up to the software to compare the PC to the various `IBREAKA/IBREAKENABLE` pairs to determine which comparison caused the exception.

The processor also provides a configurable number of data-address breakpoint registers. Each breakpoint specifies a naturally aligned power of two-sized block of bytes between one byte and 64 bytes in the processor's address space and whether the break should occur on a load or a store or both. The lowest address of the covered block of bytes is placed in one of the DBREAKA registers. The size of the covered block of bytes is placed in the low bits of the corresponding DBREAKC register while the upper two bits of the DBREAKC register contain an indication of which access types should raise the exception. The settings for each possible block size are shown in Table 4–124. The ‘x’ values under DBREAKA[i]$_{5..0}$ allow any naturally aligned address to be specified for that size. The result of other combinations of DBREAKC and DBREAKA is not defined.

Table 4–124. DBREAK Fields

Desired DBREAK Size	DBREAKC[i]5..0	DBREAKA[i]5..0
1 Byte	6’b111111	6’bxxxxxx
2 Bytes	6’b111110	6’bxxxxx0
4 Bytes	6’b111100	6’bxxxx00
8 Bytes	6’b111000	6’bxxx000
16 Bytes	6’b110000	6’bxx0000
32 Bytes	6’b100000	6’bx00000
64 Bytes	6’b000000	6’b000000

When any of the bytes accessed by a load or store matches any of the bytes of the block specified by one of the DBREAK[i] register pairs, the processor raises an exception instead of executing the load or store. Specifically, “match” is defined as:

```
(if load then DBREAKC[i]$_{30}$ else DBREAKC[i]$_{31}$) and
(DBREAKA[i] >= ($1^{26}$||DBREAKC[i]$_{5..0}$ and vAddr)) and
(DBREAKA[i] <= ($1^{26}$||DBREAKC[i]$_{5..0}$ and (vAddr+sz-1)))
```

where sz is the number of bytes in the memory access. That is, both the first and last byte of the memory access are masked by (1^{26}||DBREAKC[i]$_{5..0}$). This operation aligns both byte addresses to the DBREAK size indicated by DBREAKC[i]as in Table 4–124. If the first or last masked address or any address between them matches DBREAKA[i] then a match exists. Note that bits in DBREAKA[i]$_{5..0}$ corresponding to clear bits in DBREAKC[i]$_{5..0}$ should also be clear.

For the DBREAK exception, the DBNUM field of the DEBUGCAUSE register records, as a four bit encoded number, which of the possible DBREAK[i] registers raised the exception. If more than one DBREAK[i] matches, one of the ones that matched is recorded in DBNUM.

The processor clears IBREAKENABLE on processor reset; the IBREAKA, DBREAKA, and DBREAKC registers are undefined after reset.

4.7.6.4 Debug Exceptions

Typically DEBUGLEVEL is set to NLEVEL (highest priority for maskable interrupts) to allow debugging of other exception handlers. DEBUGLEVEL may, in certain cases be set to a lower level than NLEVEL.

The relation between the current interrupt level (CINTLEVEL, Table 4–63) and the specified debug interrupt level (DEBUGLEVEL, Table 4–120) determine whether debug interrupts can be taken. All debug exceptions (ICOUNT, IBREAK, DBREAK, BREAK, BREAK.N) are disabled when CINTLEVEL ≥ DEBUGLEVEL. In this case, the BREAK and BREAK.N instructions perform no operation.

4.7.6.5 Instruction Counting

The ICOUNT register counts instruction completions when CINTLEVEL is less than ICOUNTLEVEL. Instructions that raise an exception (including the ICOUNT exception) do not increment ICOUNT. When ICOUNT would increment to 0, it instead generates an ICOUNT exception. (See "The checkIcount Procedure" on page 203 for the formal specification.) Because ICOUNT has priority ahead of other exceptions (see Section 4.4.1.11), it is taken even if another exception would have kept the instruction from completing and, therefore, ICOUNT from incrementing.

When ICOUNTLEVEL is 1, for example, ICOUNT stops counting when an interrupt or exception occurs and starts again at the return. Neither the instruction not executed nor the return increment ICOUNT, but the re-execution of the instruction does. By this mechanism, the count of instructions can be made the same whether or not the interrupt or exception is taken. When incrementing is turned on or off by RSIL, WSR.PS, or XSR.PS instructions, the state of CINTLEVEL and ICOUNTLEVEL before the instruction begins determines whether or not the increment is done, as well as whether or not the exception is raised.

Instruction counting may be used to implement single or multi-stepping. For repeatable programs, it can also be used to determine the instruction count of the point of failure, and allow the program to be re-run up to some point before the point of failure so that the failure can be directly observed with tracing or stepping.

The purpose of the ICOUNTLEVEL register is to allow various levels of exception and interrupt processing to be visible or invisible for debugging. An ICOUNTLEVEL setting of 1 causes single-stepping to ignore exceptions and interrupts, whereas setting it to DEBUGLEVEL allows the programmer to debug exception and interrupt handlers. The ICOUNTLEVEL register should only be modified while PS.INTLEVEL or PS.EXCM is high enough that both before and after the change, ICOUNT is not incrementing.

This discussion applies for SZICOUNT=32. If SZICOUNT=2, then the upper bits appear as all ones for all purposes of reading with RSR and for comparing. In that case, WSR.ICOUNT affects only the lower two bits. The result is that the feature is really only useful for single stepping because it cannot count very far. But in other respects it behaves in the same fashion.

ICOUNTLEVEL is undefined after reset. The ICOUNT register should be read or written only when CINTLEVEL is greater than or equal to ICOUNTLEVEL, where the ICOUNT register is not incrementing (see Table 5–173).

4.7.6.6 Debug Registers

Like all special registers, the IBREAKA, IBREAKENABLE, DBREAKA, DBREAKC, and ICOUNT registers are read and written using the RSR, WSR, and XSR instructions. Figure 4–52 shows the format of the DBREAKC registers and Table 4–125 shows the DBREAKC[i] register fields.

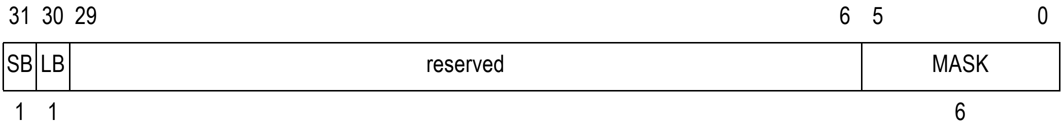


Figure 4–52. DBREAKC[i] Format

Table 4–125. DBREAKC[i] Register Fields

Field	Width (bits)	Definition
MASK	6	Mask specifying which bits of vAddr to compare to DBREAKA[i] See "Using Breakpoints" on page 199 for details.
LB	1	Load data address match enable 0 → no exception on load data address match 1 → exception on load data address match
SB	1	Store data address match enable 0 → no exception on store data address match 1 → exception on store data address match
reserved		Reserved for future use Writing a non-zero value to one of these fields results in undefined processor behavior.

4.7.6.7 Debug Interrupts

The debug data register (DDR) allows communication between a debug supervisor executing on the processor and a debugger executing on a remote host. To stop an executing program being debugged, the external debugger may use a debug interrupt. Debug interrupts share the same vector as other debug exceptions (InterruptVector[DEBUGLEVEL]), but are distinguished by the setting of the DI bit of the DEBUGCAUSE register. Both the DDR register and the debug interrupt are only available with the OCD option (see the *Tensilica On-Chip Debugging Guide*).

The INTENABLE register (see Section 4.4.4) does not contain a bit for the debug interrupt.

4.7.6.8 The checkIcount Procedure

The definition of checkIcount, used in Section 3.5.4.1 “Little-Endian Fetch Semantics” on page 29 and Section 3.5.4.2 “Big-Endian Fetch Semantics” on page 31, is:

```
procedure checkIcount ()
     if CINTLEVEL < ICOUNTLEVEL then
          if ICOUNT ≠ -1 then
               ICOUNT ← ICOUNT + 1
          elseif CINTLEVEL < DEBUGLEVEL then
               -- Exception
               DEBUGCAUSE ← 1
               EPC[DEBUGLEVEL] ← PC
               EPS[DEBUGLEVEL] ← PS
               PC ← InterruptVector[DEBUGLEVEL]
               PS.EXCM ← 1
               PS.INTLEVEL ← DEBUGLEVEL
          endif
     endif
endprocedure checkIcount
```

4.7.7 Trace Port Option

The Trace Port Option provides outputs for tracing the processor’s activity without the affect on processor timing that would happen with software profiling. For more information on this option, see the *Xtensa Microprocessor Data Book*. Because the Trace Port Option provides only additional outputs, it adds only the few architectural features listed below.

- Prerequisites: None
- Incompatible options: None

4.7.7.1 Trace Port Option Architectural Additions

Table 4–119 shows this option’s architectural additions.

Table 4–126. Trace Port Option Special Register Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
MMID	1	32	Memory Map Id	W	89

[1] Registers with a Special Register assignment are read with the RSR instruction. See Table 5–127 on page 205.

The MMID register is a write only location whose contents affect the output to the trace port and help in decoding the trace output by defining the which memory map is in force.

5. Processor State

The architectural state of an Xtensa machine consists of its AR register file, a PC, Special Registers, User Registers, TLB entries, and additional register files (added by options and designer's TIE). The Windowed Register Option causes an increase in the physical size of the AR register file but does not change the number of registers visible by instructions at any given time. To a lesser extent, caches and local memories can be considered in some ways to be architectural state. The subsections of this chapter cover each of these categories of state in turn.

The Floating-Point Coprocessor Option adds the FR register file and two User Registers called FCR and FSR. The Region Protection Option and the MMU Option add ITLB Entries and DTLB Entries. Other options add only Special Registers. Designer's TIE may add User Registers, and additional register files. Only the AR register file, the PC, and SAR are in all Xtensa processors.

Table 5–127 contains an alphabetical list of all Tensilica-defined registers that make up Xtensa processor state, including the registers added by all architectural options. The Special Register number column of most entries contains a Special Register number, which can be looked up in Section 5.3 for more information. The last column contains a reference where more information can be found in the pages following the table.

Table 5–127. Alphabetical List of Processor State

Name[1]	Description	Required Configuration Option	Special Register Number	More Detail
ACCHI	Accumulator high bits	MAC16 Option	17	Table 5–133
ACCLO	Accumulator low bits	MAC16 Option	16	Table 5–132
AR	Address registers (general registers)	Core Architecture	—	Section 5.1
ATOMCTL	Atomic Operation Control	Conditional Store Option	99	Table 5–186
BR	Boolean registers / register file	Boolean Option	4	Table 5–136
CACHEATTR	Cache attribute	XEA1 Only — see page 611	98	Table 9-250
CCOMPARE0..2	Cycle number to interrupt	Timer Interrupt Option	240-242	Table 5–176
CCOUNT	Cycle count	Timer Interrupt Option	234	Table 5–175
CPENABLE	Coprocessor enable bits	Coprocessor Option	224	Table 5–184
DBREAKA0..2	Data break address	Debug Option	144-145	Table 5–180
DBREAKC0..2	Data break control	Debug Option	160-161	Table 5–179

[1] Used in RSR, WSR, and XSR instructions.

[2] FCR & FSR are User Registers where most are system registers. These names are used in RUR and WUR instructions.

Table 5–127. Alphabetical List of Processor State (continued)

Name[1]	Description	Required Configuration Option	Special Register Number	More Detail
DEBUGCAUSE	Cause of last debug exception	Debug Option	233	Table 5–159
DDR	Debug data register	Debug Option	104	Table 5–183
DEPC	Double exception PC	Exception Option	192	Table 5–162
DTLB Entries	Data TLB entries	Region Protection Option or MMU Option	—	Section 5.5
DTLBCFG	Data TLB configuration	MMU Option	92	Table 5–152
EPC1	Level-1 exception PC	Exception Option	177	Table 5–160
EPC2..7	High level exception PC	High-Priority Interrupt Option	178-183	Table 5–161
EPS2..7	High level exception PS	High-Priority Interrupt Option	194-199	Table 5–164
EXCCAUSE	Cause of last exception	Exception Option	232	Table 5–153
EXCSAVE1	Level-1 exception save location	Exception Option	209	Table 5–166
EXCSAVE2..7	High level exception save location	High-Priority Interrupt Option	210-215	Table 5–167
EXCVADDR	Exception virtual address	Exception Option	238	Table 5–154
FCR	Floating point control register	Floating-Point Coprocessor Option	—	Table 5–189
FR	Floating point registers	Floating-Point Coprocessor Option	—	Section 5.6
FSR	Floating point status register	Floating-Point Coprocessor Option	—	Table 5–190
IBREAKA0..2	Instruction break address	Debug Option	128-129	Table 5–178
IBREAKENABLE	Instruction break enable bits	Debug Option	96	Table 5–177
ICOUNT	Instruction count	Debug Option	236	Table 5–173
ICOUNTLEVEL	Instruction count level	Debug Option	237	Table 5–174
INTCLEAR	Clear requests in INTERRUPT	Interrupt Option	227	Table 5–171
INTENABLE	Interrupt enable bits	Interrupt Option	228	Table 5–172
INTERRUPT	Interrupt request bits	Interrupt Option	226	Table 5–169
INTSET	Set Requests in INTERRUPT	Interrupt Option	226	Table 5–170
ITLB Entries	Instruction TLB entries	Region Protection Option or MMU Option	—	Section 5.5
ITLBCFG	Instruction TLB configuration	MMU Option	91	Table 5–151

[1] Used in RSR, WSR, and XSR instructions.

[2] FCR & FSR are User Registers where most are system registers. These names are used in RUR and WUR instructions.

Table 5–127. Alphabetical List of Processor State (continued)

Name[1]	Description	Required Configuration Option	Special Register Number	More Detail
LBEG	Loop-begin address	Loop Option	0	Table 5–129
LCOUNT	Loop count	Loop Option	2	Table 5–131
LEND	Loop-end address	Loop Option	1	Table 5–130
LITBASE	Literal base	Extended L32R Option	5	Table 5–137
M0..3	MAC16 data registers/register file	MAC16 Option	32-35	Table 5–134
MECR	Memory error check register	Memory ECC/Parity Option	110	Table 5–157
MEPC	Memory error PC register	Memory ECC/Parity Option	106	Table 5–163
MEPS	Memory error PS register	Memory ECC/Parity Option	107	Table 5–165
MESAVE	Memory error save register	Memory ECC/Parity Option	108	Table 5–168
MESR	Memory error status register	Memory ECC/Parity Option	109	Table 5–156
MEVADDR	Memory error virtual addr register	Memory ECC/Parity Option	111	Table 5–158
MISC0..3	Misc register 0-3	Miscellaneous Special Registers Option	244-247	Table 5–185
MMID	Memory map ID	Trace Port Option	89	Table 5–182
MR	MAC16 Data registers/register file	MAC16 Option	32-35	Table 5–134
PC	Program counter	Core Architecture	—	Section 5.2
PRID	Processor Id	Processor ID Option	235	Table 5–181
PS	Processor state	See Table 4–63 on page 87	230	Table 5–139
PTEVADDR	Page table virtual address	MMU Option	83	Table 5–149
RASID	Ring ASID values	MMU Option	90	Table 5–150
SAR	Shift-amount register	Core Architecture	3	Table 5–135
SCOMPARE1	Expected data value for S32C1I	Multiprocessor Synchronization Option	12	Table 5–138
THREADPTR	Thread pointer	Thread Pointer Option	—	Table 5–188
VECBASE	Vector Base	Relocatable Vector Option	231	Table 5–155
WindowBase	Base of current AR window	Windowed Register Option	72	Table 5–147
WindowStart	Call-window start bits	Windowed Register Option	73	Table 5–148

[1] Used in RSR, WSR, and XSR instructions.

[2] FCR & FSR are User Registers where most are system registers. These names are used in RUR and WUR instructions.

5.1 General Registers

Many Xtensa instructions operate on the general registers in the AR register file. The instructions view sixteen such registers at any given time and usually have a 4-bit specifier field in the instruction for each register they access.

These general registers are named address registers (AR) to distinguish them from the many different types of data registers that can be added to the instruction set (Section 5.6). Although the AR registers can be used to hold data as well, they are involved with both the instruction set and the execution pipeline in such a way as to make them ideally suited to contain addresses and the information used to compute addresses. They are ideally suited to computing branch conditions and targets as well, and as such fill the role of general registers in the Xtensa instruction set.

When the Windowed Register Option is enabled, there are actually more than sixteen registers in the AR register file. The windowed register ABI, described in Section 8.1, can be used in combination with the Windowed Register Option to make use of the additional registers and avoid many of the register saves and restores that would normally be associated with calls and returns. This improves both the speed and the code density of Xtensa processors.

Reads from and writes to the AR register file are always interlocked by hardware. No synchronization instructions are ever required by them.

The contents of the AR register file are undefined after reset.

5.2 Program Counter

The program counter (PC) holds the address of the next instruction to execute. It is updated by instructions as they execute. Non-branch instructions simply increment it by their length. Branch instructions, when taken, load it with a new value. Call and return instructions exist, which move values between the PC and general register AR[0]. Options such as the Loop Option change the PC in other useful ways.

Changes to and uses of the PC are always interlocked by hardware. No synchronization instructions are ever required by them.

5.3 Special Registers

Special Registers hold the majority of the state added to the processor by the Options listed in Chapter 4. Table 5–128 shows the Special Registers in numerical order with references to a more detailed description. Special Registers not listed in Table 5–128 are reserved for future use.

Table 5–128. Numerical List of Special Registers

Name[1]	Description	Required Configuration Option	Special Register Number	More Detail
LBEG	Loop-begin address	Loop Option	0	Table 5–129
LEND	Loop-end address	Loop Option	1	Table 5–130
LCOUNT	Loop count	Loop Option	2	Table 5–131
SAR	Shift-amount register	Core Architecture	3	Table 5–135
BR	Boolean registers / register file	Boolean Option	4	Table 5–136
LITBASE	Literal base	Extended L32R Option	5	Table 5–137
SCOMPARE1	Expected data value for S32C1I	Conditional Store Option	12	Table 5–138
ACCLO	Accumulator low bits	MAC16 Option	16	Table 5–132
ACCHI	Accumulator high bits	MAC16 Option	17	Table 5–133
M0..3 / MR	MAC16 data registers / register file	MAC16 Option	32-35	Table 5–134
WindowBase	Base of current AR window	Windowed Register Option	72	Table 5–147
WindowStart	Call-window start bits	Windowed Register Option	73	Table 5–148
PTEVADDR	Page table virtual address	MMU Option	83	Table 5–149
MMID	Memory map ID	Trace Port Option	89	Table 5–182
RASID	Ring ASID values	MMU Option	90	Table 5–150
ITLBCFG	Instruction TLB configuration	MMU Option	91	Table 5–151
DTLBCFG	Data TLB configuration	MMU Option	92	Table 5–152
IBREAKENABLE	Instruction break enable bits	Debug Option	96	Table 5–177
CACHEATTR	Cache attribute	XEA1 Only - see page 611	98	Table 9-250
ATOMCTL	Atomic Operation Control	Conditional Store Option	99	Table 5–186
DDR	Debug data register	Debug Option	104	Table 5–183
MEPC	Memory error PC register	Memory ECC/Parity Option	106	Table 5–163
MEPS	Memory error PS register	Memory ECC/Parity Option	107	Table 5–165
MESAVE	Memory error save register	Memory ECC/Parity Option	108	Table 5–168
MESR	Memory error status register	Memory ECC/Parity Option	109	Table 5–156
MECR	Memory error check register	Memory ECC/Parity Option	110	Table 5–157
MEVADDR	Memory error virtual addr register	Memory ECC/Parity Option	111	Table 5–158
IBREAKA0..1	Instruction break address	Debug Option	128-129	Table 5–178

[1] Used in RSR, WSR, and XSR instructions.

Table 5–128. Numerical List of Special Registers (continued)

Name[1]	Description	Required Configuration Option	Special Register Number	More Detail
`DBREAKA0..1`	Data break address	Debug Option	144-145	Table 5–180
`DBREAKC0..1`	Data break control	Debug Option	160-161	Table 5–179
`EPC1`	Level-1 exception PC	Exception Option	177	Table 5–160
`EPC2..7`	High level exception PC	High-Priority Interrupt Option	178-183	Table 5–161
`DEPC`	Double exception PC	Exception Option	192	Table 5–162
`EPS2..7`	High level exception PS	High-Priority Interrupt Option	194-199	Table 5–164
`EXCSAVE1`	Level-1 exception save location	Exception Option	209	Table 5–166
`EXCSAVE2..7`	High level exception save location	High-Priority Interrupt Option	210-215	Table 5–167
`CPENABLE`	Coprocessor enable bits	Coprocessor Option	224	Table 5–184
`INTERRUPT`	Interrupt request bits	Interrupt Option	226	Table 5–169
`INTSET`	Set requests in `INTERRUPT`	Interrupt Option	226	Table 5–170
`INTCLEAR`	Clear requests in `INTERRUPT`	Interrupt Option	227	Table 5–171
`INTENABLE`	Interrupt enable bits	Interrupt Option	228	Table 5–172
`PS`	Processor state	See Table 4–63 on page 87	230	Table 5–139
`VECBASE`	Vector Base	Relocatable Vector Option	231	Table 5–155
`EXCCAUSE`	Cause of last exception	Exception Option	232	Table 5–153
`DEBUGCAUSE`	Cause of last debug exception	Debug Option	233	Table 5–159
`CCOUNT`	Cycle count	Timer Interrupt Option	234	Table 5–175
`PRID`	Processor Id	Processor ID Option	235	Table 5–181
`ICOUNT`	Instruction count	Debug Option	236	Table 5–173
`ICOUNTLEVEL`	Instruction count level	Debug Option	237	Table 5–174
`EXCVADDR`	Exception virtual address	Exception Option	238	Table 5–154
`CCOMPARE0..2`	Cycle number to generate interrupt	Timer Interrupt Option	240-242	Table 5–176
`MISC0..3`	Misc register 0-3	Miscellaneous Special Registers Option	244-247	Table 5–185

[1] Used in `RSR`, `WSR`, and `XSR` instructions.

Section 5.3.1 describes the process of reading and writing these special registers, while the sections that follow describe groups of specific Special Registers in more detail. A table is included for each special register, which includes information specific to that special register. The gray shaded rows describe the information that is contained in the unshaded rows immediately below them.

The first row shows the Special Register number, the Name (which is used in the RSR.*, WSR.*, and XSR.* instruction names), a short description, and the value immediately after reset.

The second row shows the Option that creates the Special Register, the count or number of such special registers, the number of bits in the special register, whether access to the register is privileged (requires CRING=0) or not, and whether XSR.* is a legal instruction or not. The Option that creates the Special Register is described in Chapter 4 including more information on each Special Register.

The third row shows the function of the WSR.* and RSR.* instructions for this Special Register. The function of the XSR.* instruction is the combination of the RSR.* and the WSR.* instructions.

The fourth row shows the other instructions that affect or are affected by this Special Register.

The last row of each Special Register's table shows what SYNC instructions are required when using this Special Register. If no SYNC instructions are ever required, the row is left out. On the left is an instruction or other action that changes the value of the Special Register. On the right is an instruction or other action that makes use of the value of the Special Register. If a SYNC instruction is required for this pair of operations to work as they should, it is listed in the middle. Wherever a DSYNC is required an ISYNC, RSYNC, or ESYNC can also be used. Wherever an ESYNC is required an ISYNC or RSYNC can also be used. Wherever an RSYNC is required an ISYNC can also be used. Note that the 16-bit versions (*.N) of 24-bit instructions are not listed separately but always have exactly the same requirements. Versions T1050 and before required additional SYNC instructions in some cases as described in Section A.8 on page 621.

Because of the importance of its subfields, the PS Special Register is a special case. Its subfields are listed in the same format as special registers. The synchronizations needed simply because the register has been written are listed under the entire register, while the synchronizations needed because the value of a subfield has been changed are listed under the subfield.

5.3.1 Reading and Writing Special Registers

The RSR.*, WSR.*, and XSR.* instructions access the special registers. The accesses to the Special Registers act as separate instructions in many ways. For the full instruction name, replace the '*' in the instructions with the name as given in the Special Register Tables in this section.

Each RSR.* instruction moves a value from a Special Register to a general (AR) register. Each WSR.* instruction moves a value from a general (AR) register to a Special Register. Each XSR.* instruction exchanges the values in a general (AR) register and a Special Register. Some Special Registers do not allow this exchange. The Special Register tables in this section show which do and do not allow this exchange. The exchange takes place with the two reads taking place first, and then the two writes. In some cases, the write of a Special Register can affect other behavior of the processor. In general, this behavior change does not occur until after the instruction (including `XSR.*`) has completed execution.

Some of the Special Registers have interactions with other instructions or with hardware execution. These interactions are also listed in the Special Register tables in this section. Because modification of many Special Registers is an unusual occurrence, synchronization instructions are used to ensure that their values have propagated everywhere before certain other actions are allowed to take place. Some of the interlocks would be costly in performance or in gates if done in hardware, and the synchronization instructions can be the most efficient solution.

5.3.2 LOOP Special Registers

The Loop Option adds the three registers shown in Table 5–129 through Table 5–131 for controlling zero overhead loops. When the `PC` reaches `LEND`, it executes at `LBEG` instead and decrements `LCOUNT`. When `LCOUNT` reaches zero, the loop back does not occur.

Table 5–129. `LBEG` - Special Register #0

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>0</td><td>LBEG</td><td colspan="3">Loop begin - address of beginning of zero overhead loop</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Loop Option</td><td>1</td><td>32</td><td>No</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">LBEG ← AR[t]</td><td colspan="3">AR[t] ← LBEG</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">LOOP/LOOPGTZ/LOOPNEZ</td><td colspan="3">Branch at end of zero overhead loop</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR LBEG ⇒ ISYNC ⇒ Potential branch caused by attempt to execute LEND</td></tr>
</table>

Table 5–130. **LEND – Special Register #1**

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>1</td><td>LEND</td><td colspan="3">Loop end - address of instruction after zero overhead loop</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Loop Option</td><td>1</td><td>32</td><td>No</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">LEND ← AR[t]</td><td colspan="3">AR[t] ← LEND</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">LOOP/LOOPGTZ/LOOPNEZ</td><td colspan="3">Branch at end of zero overhead loop</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR LEND ⇒ ISYNC ⇒ Potential branch caused by attempt to execute LEND</td></tr>
</table>

Table 5–131. **LCOUNT - Special Register #2**

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>2</td><td>LCOUNT</td><td colspan="3">Loop count remaining</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Loop Option</td><td>1</td><td>32</td><td>No</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">LCOUNT ← AR[t]</td><td colspan="3">AR[t] ← LCOUNT</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">LOOP/LOOPGTZ/LOOPNEZ</td><td colspan="3">Branch at end of zero overhead loop</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR LCOUNT ⇒ ESYNC ⇒ RSR/XSR LCOUNT</td></tr>
<tr><td colspan="6">WSR/XSR LCOUNT ⇒ ISYNC ⇒ Potential branch caused by attempt to execute LEND</td></tr>
<tr><td colspan="6">WSR/XSR LCOUNT to zero⇒ ISYNC ⇒ WSR/XSR PS.EXCM with zero (for protection)</td></tr>
</table>

5.3.3 MAC16 Special Registers

The MAC16 Option adds the six registers described in Table 5–132 through Table 5–134.

Table 5–132. ACCLO – Special Register #16

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>16</td><td>ACCLO</td><td colspan="3">Accumulator - low bits</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">MAC16 Option</td><td>1</td><td>32</td><td>No</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">ACC$_{31..0}$ ← AR[t]</td><td colspan="3">AR[t] ← ACC$_{31..0}$</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">MUL.*/MULA.*/MULS.*/UMUL.*</td><td colspan="3">MULA.*/MULS.*</td></tr>
</table>

Table 5–133. ACCHI – Special Register #17

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>17</td><td>ACCHI</td><td colspan="3">Accumulator - high bits</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">MAC16 Option</td><td>1</td><td>8</td><td>No</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">ACC$_{39..32}$ ← AR[t]$_{7..0}$
Undefined if AR[t]$_{31..8}$ ≠ AR[t]$_{7}^{24}$</td><td colspan="3">AR[t] ← ACC$_{39}^{24}$||ACC$_{39..32}$</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">MUL.*/MULA.*/MULS.*/UMUL.*</td><td colspan="3">MULA.*/MULS.*</td></tr>
</table>

Table 5–134. M0..3 – Special Register #32-35

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>32-35</td><td>M0..3 / MR[1]</td><td colspan="3">MAC16 data registers / register file[1]</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">MAC16 Option</td><td>4</td><td>32</td><td>No</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">M[sr$_{1..0}$] ← AR[t]</td><td colspan="3">AR[t] ← M[sr$_{1..0}$]</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">LDDEC/LDINC/MULA*.LDDEC/MULA*.LDINC</td><td colspan="3">MUL.*D*/MULA.*D*/MULS.*D*</td></tr>
</table>

[1] These registers are known as MR[0..3] in hardware and as m0..3 in the software.

5.3.4 Other Unprivileged Special Registers

The SAR Special Register is included in the Xtensa Core Architecture, while the BR, LITBASE, and SCOMPARE1 Special Registers are added by the options shown along with other information about them in Table 5–135 through Table 5–138.

Table 5–135. SAR - Special Register #3

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>3</td><td>SAR</td><td colspan="3">Shift amount register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Core Architecture (see page 25)</td><td>1</td><td>6</td><td>No</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">SAR ← AR[t]$_{5..0}$
Undefined if AR[t]$_{31..6}$ ≠ 0^{26}</td><td colspan="3">AR[t] ← 0^{26}||SAR</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">SSL/SSR/SSAI/SSA8B/SSA8L</td><td colspan="3">SLL/SRL/SRA/SRC</td></tr>
</table>

Table 5–136. BR - Special Register #4

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>4</td><td>BR / b0..15[1]</td><td colspan="3">Boolean register / register file[1]</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Boolean Option</td><td>1</td><td>16</td><td>No</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">BR ← AR[t]$_{15..0}$
Undefined if AR[t]$_{31..16}$ ≠ 0^{16}</td><td colspan="3">AR[t] ← 0^{16}||BR</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">ALL4/ALL8/ANDB/ANDBC/ANY4/ANY8/ ORB/ORBC/XORB/OEQ.S/OLE.S/OLT.S/ UEQ.S/ULE.S/ULT.S/UN.S/User TIE</td><td colspan="3">ALL4/ALL8/ANDB/ANDBC/ANY4/ANY8/ ORB/ORBC/XORB/ BF/BT/MOVF/MOVF.S/MOVT/MOVT.S</td></tr>
</table>

[1] This register is known as Special Register BR or as individual Boolean bits b0..15.

Table 5–137. `LITBASE` - Special Register #5

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>5</td><td>LITBASE</td><td colspan="3">Literal base register</td><td>bit-0 clear[1]</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Extended L32R Option</td><td>1</td><td>21</td><td>No</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">$\mathtt{LITBASE} \leftarrow \mathtt{AR[t]}_{31..12} \| 0^{11} \| \mathtt{AR[t]}_{0}$
Undefined if $\mathtt{AR[t]}_{11..1} \neq 0^{11}$</td><td colspan="3">$\mathtt{AR[t]} \leftarrow \mathtt{LITBASE}_{31..12} \| 0^{11} \| \mathtt{LITBASE}_{0}$</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">L32R</td></tr>
<tr><td colspan="6">[1] After reset bit-0 is clear but the remainder of the register is undefined.</td></tr>
</table>

Table 5–138. `SCOMPARE1` - Special Register #12

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>12</td><td>SCOMPARE1</td><td colspan="3">Comparison register for the S32C1I instruction</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Conditional Store Option</td><td>1</td><td>32</td><td>No</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">SCOMPARE1 ← AR[t]</td><td colspan="3">AR[t] ← SCOMPARE1</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">S32C1I</td></tr>
</table>

5.3.5 Processor Status Special Register

The Processor Status Special Register is made up of multiple fields with different purposes within the processor. They are combined into one register to simplify the saving and restoring of state for exceptions, interrupts, and context switches. Table 5–139 describes the register as a whole, while Table 5–140 through Table 5–146 describe the individual pieces of the register in a similar format.

The synchronization section of Table 5–139 gives requirements that must be met whenever the `PS` register is written regardless of whether any of its bits are changed. The synchronization sections of Table 5–140 through Table 5–146 give requirements that must be met only if that portion of the PS register is being modified.

Table 5–139. PS – Special Register #230

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>230</td><td>PS</td><td colspan="3">Miscellaneous program state</td><td>0x10 or 0x1F[1]</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Exception Option</td><td>1</td><td>15</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">PS ← 0^{13}||AR[t]$_{18..16}$||0^{4}||AR[t]$_{11..0}$
PS.RING should be changed only when CEXCM=1 before the instruction making the change.</td><td colspan="3">AR[t] ← PS</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">CALL[X]4-12/RFE/RFDO/RFDD/RFWO/RFWU/RFI RSIL/WAITI/interrupts/exceptions</td><td colspan="3">CALL[X]4-12/ENTRY/RETW/interrupts/loop-back Privileged-instructions/ld-st-instructions/exceptions</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">See following entries for subfields of PS. Write to PS.X means a write to PS that changes subfield X.</td></tr>
</table>

[1] PS is 5'h1F after reset if the.Interrupt Option is configured but reads as 5'h10 if it is not.

Table 5–140. PS.INTLEVEL – Special Register #230 (part)

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>230 Part</td><td>PS.INTLEVEL</td><td colspan="3">Interrupt level mask part of PS (Table 5–139)</td><td>0x0 or 0xF[1]</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Interrupt Option</td><td>1</td><td>4</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">(see Table 5–139)</td><td colspan="3">(see Table 5–139)</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">RFI/RFDD/RFDO/RSIL/WAITI/ Hi-level-interrupts/debug-exceptions/NMI</td><td colspan="3">RSIL/interrupts/debug-exceptions</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">Write to PS.INTLEVEL is a write to PS that changes subfield INTLEVEL.
WSR/XSR PS.INTLEVEL ⇒ RSYNC ⇒ Change in accepting interrupts
If PS.EXCM and PS.INTLEVEL are both changed in the same WSR.PS or XSR.PS instruction in such a way that a particular interrupt is forbidden both before and after the instruction, there will be no cycle during the instruction where the interrupt may be taken. Thus PS.EXCM may be cleared and PS.INTLEVEL raised (or PS.EXCM set and PS.INTLEVEL lowered) in the same instruction and no gap is opened between them.
WSR/XSR PS.INTLEVEL ⇒ DSYNC ⇒ Change in taking debug exception (interrupt level)
RFI/RFDD/RFDO/RSIL/WAITI ⇒ (none) ⇒ RSIL or change in accepting interrupts/debug-exceptions
Hi-level-interrupts/debug-excep/NMI ⇒ (none) ⇒ RSIL or change in accepting interrupts/debug-exceptions</td></tr>
</table>

[1] PS.INTLEVEL is 4'hF after reset if the.Interrupt Option is configured but reads as 4'h0 if it is not.

Table 5–141. PS.EXCM – Special Register #230 (part)

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>230 Part</td><td>PS.EXCM</td><td colspan="3">Exception mask part of PS (Table 5–139)</td><td>0x1</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Exception Option</td><td>1</td><td>1</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">(see Table 5–139)</td><td colspan="3">(see Table 5–139)</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">RFI/RFDD/RFDO/RFE/RFWO/RFWU
interrupts/exceptions</td><td colspan="3">CALL[X]4-12/ENTRY/RETW/interrupts/loop-back
Ifetch/privileged-instr/ld-st-instructions/exceptions</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">Write to PS.EXCM is a write to PS that changes subfield EXCM.
WSR/XSR PS.EXCM ⇒ ISYNC ⇒ Changes in instruction fetch privilege
WSR/XSR PS.EXCM ⇒ RSYNC ⇒ Change in accepting Interrupts
If PS.EXCM and PS.INTLEVEL are both changed in the same WSR.PS or XSR.PS instruction in such a way that a particular interrupt is forbidden both before and after the instruction, there will be no cycle during the instruction where the interrupt may be taken. Thus PS.EXCM may be cleared and PS.INTLEVEL raised (or PS.EXCM set and PS.INTLEVEL lowered) in the same instruction without a gap in interrupt masking.
WSR/XSR PS.EXCM to one ⇒ (none) ⇒ Restore non-zero LCOUNT value
WSR/XSR LCOUNT to zero ⇒ ISYNC ⇒ WSR/XSR PS.EXCM with zero (for protection)
WSR/XSR PS.EXCM ⇒ ESYNC ⇒ CALL[X]4-12/ENTRY/RETW
Note: In the Windowed Register Option, any instruction with an AR register operand can cause overflow exceptions.
WSR/XSR PS.EXCM ⇒ DSYNC ⇒ Changes in data fetch privilege
WSR/XSR PS.EXCM ⇒ (none) ⇒ Double exception vector or not
RFI/RFDD/RFDO/RFE ⇒ (none) ⇒ Anything
RFWO/RFWU ⇒ (none) ⇒ Anything
Interrupts/exceptions⇒ (none) ⇒ Anything</td></tr>
</table>

Table 5–142. `PS.UM` – Special Register #230 (part)

SR#	Name	Description	Reset Value
`230` Part	`PS.UM`	User vector mode part of `PS` (Table 5–139)	0x`0`

Option	Count	Bits	Privileged?	XSR Legal?
Exception Option	1	1	Yes	Yes

WSR Function	RSR Function
(see Table 5–139)	(see Table 5–139)

Other Changes to the Register	Other Effects of the Register
`RFI/RFDD/RFDO`	`RSIL`/level-1-interrupts general-exceptionsdebug-exceptions

Instruction ⇒ `xSYNC` ⇒ Instruction
Write to `PS.UM` is a write to `PS` that changes subfield `UM`.
`WSR/XSR PS.UM` ⇒ `RSYNC` ⇒ Level-1-interrupts/general-exceptions/debug-exceptions
Note: In the Windowed Register Option, any instruction with an `AR` register operand can cause overflow exceptions.

Table 5–143. `PS.RING` – Special Register #230 (part)

SR#	Name	Description	Reset Value
`230` Part	`PS.RING`	Ring part of `PS` (Table 5–139)	0x`0`

Option	Count	Bits	Privileged?	XSR Legal?
MMU Option	1	2	Yes	Yes

WSR Function	RSR Function
(see Table 5–139)	(see Table 5–139)

Other Changes to the Register	Other Effects of the Register
`RFI/RFDD/RFDO`	Hi-level-interrupts/debug-exception/ Privileged-instructions/ld-st-instructions

Instruction ⇒ `xSYNC` ⇒ Instruction
Write to `PS.RING` is a write to `PS` that changes subfield `RING`.
`WSR/XSR PS.RING` ⇒ `ISYNC` ⇒ Changes in instruction fetch privilege
`WSR/XSR PS.RING` ⇒ `DSYNC` ⇒ Changes in data fetch privilege

Table 5–144. PS.OWB – Special Register #230 (part)

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>230 Part</td><td>PS.OWB</td><td colspan="3">Old window base part of PS (Table 5–139)</td><td>0x0</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Windowed Register Option</td><td>1</td><td>4</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">(see Table 5–139)</td><td colspan="3">(see Table 5–139)</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">RFI/RFDD/RFDO/overflow-or-underflow-exception</td><td colspan="3">RFWO/RFWU/RSIL/hi-level-interrupt/debug-exception</td></tr>
</table>

Table 5–145. PS.CALLINC – Special Register #230 (part)

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>230 Part</td><td>PS.CALLINC</td><td colspan="3">Call increment part of PS (Table 5–139)</td><td>0x0</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Windowed Register Option</td><td>1</td><td>2</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">(see Table 5–139)</td><td colspan="3">(see Table 5–139)</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">CALL[X]4-12/RFI/RFDD/RFDO</td><td colspan="3">ENTRY/RSIL/hi-level-interrupt/debug-exception</td></tr>
</table>

Table 5–146. PS.WOE – Special Register #230 (part)

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>230 Part</td><td>PS.WOE</td><td colspan="3">Window overflow enable part of PS (Table 5–139)</td><td>0x0</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Windowed Register Option</td><td>1</td><td>1</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">(see Table 5–139)</td><td colspan="3">(see Table 5–139)</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">RFI/RFDD/RFDO</td><td colspan="3">CALL4-12/CALLX4-12/ENTRY/RETW/RSIL/ Hi-level-interrupt/debug-exception/overflow-exception</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">Write to PS.WOE is a write to PS that changes subfield WOE.
WSR/XSR PS.WOE ⇒ RSYNC ⇒ CALL4-12/CALLX4-12/ENTRY/RETW
WSR/XSR PS.WOE ⇒ RSYNC ⇒ Overflow-exception
Note: In the Windowed Register Option, any instruction with an AR register operand can cause overflow exceptions.</td></tr>
</table>

5.3.6 Windowed Register Option Special Registers

The Windowed Register Option Special registers are described in Table 5–147 and Table 5–148.

Table 5–147. `WindowBase` - Special Register #72

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>72</td><td>WindowBase</td><td colspan="3">Base of current AR register window</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Windowed Register Option</td><td>1</td><td>log2(NAREG/4)</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">WindowBase ← $\mathrm{AR[t]}_{X-1..0}$
Undefined if $\mathrm{AR[t]}_{31..X} \neq 0^{32-X}$
X = log2(NAREG/4)</td><td colspan="3">AR[t] ← 0^{32-X}||WindowBase
X = log2(NAREG/4)</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">ENTRY/MOVSP/RETW/RFW*/ROTW
Overflow/underflow-exception</td><td colspan="3">Any instruction which accesses the AR register file</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR WINDOWBASE ⇒ RSYNC ⇒ Any use or def of an ARregister</td></tr>
</table>

Table 5–148. `WindowStart` - Special Register #73

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>73</td><td>WindowStart</td><td colspan="3">Call-window start bits</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Windowed Register Option</td><td>1</td><td>NAREG/4</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">WindowStart ← $\mathrm{AR[t]}_{NAREG/4-1..0}$
Undefined if $\mathrm{AR[t]}_{31..NAREG/4} \neq 0^{32-NAREG/4}$</td><td colspan="3">AR[t] ← $0^{32-NAREG/4}$||WindowStart</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">ENTRY/MOVSP/RETW/RFWO/RFWU</td><td colspan="3">Any instruction which accesses the AR register file</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR WINDOWSTART ⇒ RSYNC ⇒ Any use of an AR register when CWOE=1
WSR/XSR WINDOWSTART ⇒ RSYNC ⇒ Any def of an AR register when CWOE=1</td></tr>
</table>

5.3.7 Memory Management Special Registers

The Special Registers for managing memory are described in Table 5–149 through Table 5–152.

Table 5–149. `PTEVADDR` – Special Register #83

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>83</td><td>PTEVADDR</td><td colspan="3">Virtual address for page table lookups</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">MMU Option</td><td>1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">$\mathtt{PTEVADDR}_{\mathtt{VABITS-1..X}} \leftarrow \mathtt{AR[t]}_{\mathtt{VABITS-1..X}}$
$\mathtt{X} = \mathtt{VABITS+log2(PTEbytes)-min(PTEPageSizes)}$</td><td colspan="3">$\mathtt{AR[t]} \leftarrow \mathtt{PTEVADDR}_{\mathtt{VABITS-1..Y}} \| 0^{\mathtt{Y}}$
$\mathtt{Y} = \mathtt{log2(PTEbytes)}$</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">Any instruction/data address translation</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR PTEVADDR ⇒ ISYNC ⇒ Any instruction access that might miss the ITLB</td></tr>
<tr><td colspan="6">WSR/XSR PTEVADDR ⇒ DSYNC ⇒ Any load/store access that might miss the DTLB</td></tr>
</table>

Table 5–150. `RASID` – Special Register #90

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>90</td><td>RASID</td><td colspan="3">Current ASID values for each protection ring</td><td>0x04030201</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">MMU Option</td><td>1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">$\mathtt{RASID} \leftarrow \mathtt{AR[t]}_{\mathtt{31..8}} \| 0^{7} \| 1^{1}$</td><td colspan="3">$\mathtt{AR[t]} \leftarrow \mathtt{RASID}$</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">Any instruction/data address translation</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR RASID ⇒ ISYNC ⇒ Instruction address translation that depends on the change</td></tr>
<tr><td colspan="6">WSR/XSR RASID ⇒ DSYNC ⇒ Data address translation that depends on the change</td></tr>
</table>

Table 5–151. ITLBCFG – Special Register #91

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>91</td><td>ITLBCFG</td><td colspan="3">Instruction TLB configuration</td><td>0x00000000</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">MMU Option</td><td>1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">ITLBCFG ← AR[t]
Affected ways should be invalidated after change.</td><td colspan="3">AR[t] ← ITLBCFG</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">Any instruction address translation</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR ITLBCFG ⇒ ISYNC ⇒ Instruction address translation that depends on the change</td></tr>
</table>

Table 5–152. DTLBCFG – Special Register #92

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>92</td><td>DTLBCFG</td><td colspan="3">Data TLB configuration</td><td>0x00000000</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">MMU Option</td><td>1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">DTLBCFG ← AR[t]
Affected ways should be invalidated after change.</td><td colspan="3">AR[t] ← DTLBCFG</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">Any data address translation</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR DTLBCFG ⇒ DSYNC ⇒ Any data address translation that depends on the change</td></tr>
</table>

5.3.8 Exception Support Special Registers

The Special Registers that provide information for the handling of an exception are described in Table 5–153 through Table 5–159.

Table 5–153. EXCCAUSE – Special Register #232

SR#	Name	Description	Reset Value
232	EXCCAUSE	Exception cause register	Undefined

Option	Count	Bits	Privileged?	XSR Legal?
Exception Option	1	6	Yes	Yes

WSR Function	RSR Function
$\mathtt{EXCCAUSE} \leftarrow \mathtt{AR[t]}_{5..0}$ Undefined if $\mathtt{AR[t]}_{31..6} \neq 0^{26}$	$\mathtt{AR[t]} \leftarrow 0^{26} \Vert \mathtt{EXCCAUSE}$

Other Changes to the Register	Other Effects of the Register
Exception or interrupt	

Table 5–154. EXCVADDR – Special Register #238

SR#	Name	Description	Reset Value
238	EXCVADDR	Exception virtual address register	Undefined

Option	Count	Bits	Privileged?	XSR Legal?
Exception Option	1	32	Yes	Yes

WSR Function	RSR Function
$\mathtt{EXCVADDR} \leftarrow \mathtt{AR[t]}$	$\mathtt{AR[t]} \leftarrow \mathtt{EXCVADDR}$ AR[t] is undefined if $\mathtt{CEXCM} = 0$

Other Changes to the Register	Other Effects of the Register
Some exceptions (see Table 4–64 on page 89), hardware table walk (see Section 4.6.5.9 on page 174)	

Table 5–155. VECBASE – Special Register #231

SR#	Name	Description	Reset Value
231	VECBASE	Vector Base	User Defined[1]

Option	Count	Bits	Privileged?	XSR Legal?
Relocatable Vector Option	1	32	Yes	Yes

WSR Function	RSR Function
$\mathtt{VECBASE} \leftarrow \mathtt{AR[t]}$	$\mathtt{AR[t]} \leftarrow \mathtt{VECBASE}$

Other Changes to the Register	Other Effects of the Register
	Exception Vector Locations

[1] The reset value of VECBASE is set by the user as part of the configuration

Table 5–156. MESR – Special Register #109

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>109</td><td>MESR</td><td colspan="3">Memory error status register</td><td>32’hXXXX0C00</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Memory ECC/Parity Option</td><td>1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">MESR ← AR[t]</td><td colspan="3">AR[t] ← MESR</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">Memoryerror-exception, memory error without exception</td><td colspan="3">Controls memory error logic</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR MESR ⇒ ISYNC ⇒ Change in error behavior on instruction memories</td></tr>
<tr><td colspan="6">WSR/XSR MESR ⇒ DSYNC ⇒ Change in error behavior on data memories</td></tr>
</table>

Table 5–157. MECR – Special Register #110

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>110</td><td>MECR</td><td colspan="3">Memory error check register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Memory ECC/Parity Option</td><td>1</td><td>22</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">MECR ← AR[t]</td><td colspan="3">AR[t] ← MECR</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">Memoryerror-exception, memory error without exception, Loads when MESR[9] is set.</td><td colspan="3">Stores when MESR[9] is set.</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR MECR ⇒ ISYNC ⇒ Check bit write to instruction memories</td></tr>
<tr><td colspan="6">WSR/XSR MECR ⇒ DSYNC ⇒ Check bit write to data memories</td></tr>
</table>

Table 5–158. MEVADDR – Special Register #111

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>111</td><td>MEVADDR</td><td colspan="3">Memory error virtual address register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Memory ECC/Parity Option</td><td>1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">MEVADDR ← AR[t]</td><td colspan="3">AR[t] ← MEVADDR</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">Memoryerror-exception, memory error without exception</td><td colspan="3"></td></tr>
</table>

Table 5–159. **DEBUGCAUSE - Special Register #233**

SR#	Name	Description			Reset Value
233	DEBUGCAUSE	Debug cause register			Undefined
Option		**Count**	**Bits**	**Privileged?**	**XSR Legal?**
Debug Option		1	12	Yes	No
WSR Function			**RSR Function**		
Reserved			AR[t] ← 0^{20}‖DEBUGCAUSE		
Other Changes to the Register			**Other Effects of the Register**		
Debug exception or interrupt					

5.3.9 Exception State Special Registers

The Special Registers that save the PC and PS values and an initial register value for each of the levels are described in Table 5–160 through Table 5–162.

Table 5–160. **EPC1 - Special Register #177**

SR#	Name	Description			Reset Value
177	EPC1	Exception PC[1]			Undefined
Option		**Count**	**Bits**	**Privileged?**	**XSR Legal?**
Exception Option		1	32	Yes	Yes
WSR Function			**RSR Function**		
EPC[1] ← AR[t]			AR[t] ← EPC[1]		
Other Changes to the Register			**Other Effects of the Register**		
General-exception/overflow-or-underflow-exception			RFE/RFWO/RFWU		

Table 5–161. **EPC2..7 - Special Register #178-183**

SR#	Name	Description			Reset Value
178-183	EPC2..7	Exception PC[2..7]			Undefined
Option		**Count**	**Bits**	**Privileged?**	**XSR Legal?**
High-Priority Interrupt Option		NLEVEL +NNMI-1	32	Yes	Yes
WSR Function			**RSR Function**		
EPC[$sr_{3..0}$] ← AR[t]			AR[t] ← EPC[$sr_{3..0}$] AR[t] is undefined if $sr_{3..0}$ > NLEVEL+NNMI		
Other Changes to the Register			**Other Effects of the Register**		
Level[$sr_{3..0}$]–Interrupt/debug-exception/NMI			RFI[$sr_{3..0}$]/RFDO/RFDD		

Table 5–162. DEPC – Special Register #192

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>192</td><td>DEPC</td><td colspan="3">Double exception PC</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Exception Option</td><td>1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">DEPC ← AR[t]</td><td colspan="3">AR[t] ← DEPC</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">Double exception</td><td colspan="3">RFDE</td></tr>
</table>

Table 5–163. MEPC – Special Register #106

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>106</td><td>MEPC</td><td colspan="3">Memory error PC register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Memory ECC/Parity Option</td><td>1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">MEPC ← AR[t]</td><td colspan="3">AR[t] ← MEPC
AR[t] is undefined unless MESR[0] is set.</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">Memoryerror-exception</td><td colspan="3">RFME</td></tr>
</table>

Table 5–164. EPS2..7 – Special Register #194-199

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>194-199</td><td>EPS2..7</td><td colspan="3">Exception processor status register[2..7]</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">High-Priority Interrupt Option</td><td>NLEVEL
+NNMI-1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">EPS[sr$_{3..0}$] ← AR[t]</td><td colspan="3">AR[t] ← EPS[sr$_{3..0}$]
AR[t] is undefined if sr$_{3..0}$ > NLEVEL+NNMI</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">Level[sr$_{3..0}$]–Interrupt/debug-exception/NMI</td><td colspan="3">RFI[sr$_{3..0}$]/RFDO/RFDD</td></tr>
</table>

Table 5–165. **MEPS** – Special Register #107

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>107</td><td>MEPS</td><td colspan="3">Memory error PS register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Memory ECC/Parity Option</td><td>1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">MEPS ← AR[t]</td><td colspan="3">AR[t] ← MEPS
AR[t] is undefined unless MESR[0] is set.</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">Memoryerror-exception</td><td colspan="3">RFME</td></tr>
</table>

Table 5–166. **EXCSAVE1** – Special Register #192

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>192</td><td>EXCSAVE1</td><td colspan="3">Exception save register [1]</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Exception Option</td><td>1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">EXCSAVE[1] ← AR[t]</td><td colspan="3">AR[t] ← EXCSAVE[1]</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3"></td></tr>
</table>

Table 5–167. **EXCSAVE2..7**- Special Register #210-215

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>210-215</td><td>EXCSAVE2..7</td><td colspan="3">Exception save register [2..7]</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">High-Priority Interrupt Option</td><td>NLEVEL
+NNMI-1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">EXCSAVE[$sr_{3..0}$] ← AR[t]</td><td colspan="3">AR[t] ← EXCSAVE[$sr_{3..0}$]
AR[t] is undefined if $sr_{3..0}$ > NLEVEL+NNMI</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3"></td></tr>
</table>

Table 5–168. MESAVE- Special Register #108

SR#	Name	Description	Reset Value
109	MESAVE	Memory error save register	Undefined

Option	Count	Bits	Privileged?	XSR Legal?
Memory ECC/Parity Option	1	32	Yes	Yes

WSR Function	RSR Function
MESAVE ← AR[t]	AR[t] ← MESAVE

Other Changes to the Register	Other Effects of the Register

5.3.10 Interrupt Special Registers

The Special Registers that manage interrupt handling are described in Table 5–169 through Table 5–172.

Table 5–169. INTERRUPT - Special Register #226 (read)

SR#	Name	Description	Reset Value
226	INTERRUPT	Interrupt pending register	Undefined

Option	Count	Bits	Privileged?	XSR Legal?
Interrupt Option	1	NINTERRUPT	Yes	No

WSR Function	RSR Function
see Table 5–170 and Table 5–171	AR[t] ← $0^{32-\text{NINTERRUPT}} \Vert \text{INTERRUPT}$

Other Changes to the Register	Other Effects of the Register
Assertion/deassertion of interrupt signals/ WSR.CCOMPAREn	Pipeline takes interrupt

Instruction	⇒	xSYNC	⇒	Instruction
WSR INTSET	⇒	ESYNC	⇒	RSR INTERRUPT
WSR INTCLEAR	⇒	ESYNC	⇒	RSR INTERRUPT

Table 5–170. INTSET – Special Register #226 (write)

<table>
<tr><th>SR#</th><th>Name</th><th colspan="2">Description</th><th>Reset Value</th></tr>
<tr><td>226</td><td>INTSET</td><td colspan="2">Interrupt set register</td><td>No separate state</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Interrupt Option</td><td>1</td><td>NINTERRUPT</td><td>Yes</td><td>No</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">INTERRUPT ← INTERRUPT or AR[t]$_{X-1..0}$
Undefined if AR[t]$_{31..X} \neq 0^{32-X}$
X = NINTERRUPT
Only software interrupt bits can be set.</td><td colspan="3">see Table 5–169</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">(State is INTERRUPT)</td><td colspan="3">(State is INTERRUPT)</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR INTSET ⇒ ESYNC ⇒ RSR INTERRUPT
WSR INTSET⇒ RSYNC ⇒ Instruction which must execute after the software interrupt</td></tr>
</table>

Table 5–171. INTCLEAR – Special Register #227

<table>
<tr><th>SR#</th><th>Name</th><th colspan="2">Description</th><th>Reset Value</th></tr>
<tr><td>227</td><td>INTCLEAR</td><td colspan="2">Interrupt clear register</td><td>No separate state</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Interrupt Option</td><td>1</td><td>NINTERRUPT</td><td>Yes</td><td>No</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">INTERRUPT ← INTERRUPT and not AR[t]$_{X-1..0}$
Undefined if AR[t]$_{31..X} \neq 0^{32-X}$
X = NINTERRUPT
Bits in AR[t]$_{X-1..0}$ may be set without causing harm.
Only bits which can be cleared by this write are affected.</td><td colspan="3">AR[t] ← undefined^{32}</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">(State is INTERRUPT)</td><td colspan="3">(State is INTERRUPT)</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR INTCLEAR ⇒ ESYNC ⇒ RSR INTERRUPT
WSR INTCLEAR⇒ RSYNC ⇒ Instruction which must execute after the cleared interrupt</td></tr>
</table>

Table 5–172. INTENABLE - Special Register #228

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>228</td><td>INTENABLE</td><td colspan="3">Interrupt enable register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Interrupt Option</td><td>1</td><td>NINTERRUPT</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">$\texttt{INTENABLE} \leftarrow \texttt{AR[t]}_{\texttt{NINTERRUPT-1..0}}$
Undefined if $\texttt{AR[t]}_{\texttt{31..X}} \neq 0^{\texttt{32-X}}$
$\texttt{X = NINTERRUPT}$</td><td colspan="3">$\texttt{AR[t]} \leftarrow 0^{\texttt{32-NINTERRUPT}} \| \texttt{INTENABLE}$</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">Pipeline takes interrupt</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR INTENABLE ⇒ ESYNC ⇒ RSR/XSR INTENABLE</td></tr>
<tr><td colspan="6">WSR/XSR INTENABLE⇒ RSYNC ⇒ Any instruction which must wait for INTENABLE changes</td></tr>
</table>

5.3.11 Timing Special Registers

The Special Registers that manage instruction counting and cycle counting, including timer interrupts are described in Table 5–173 through Table 5–176.

Table 5–173. ICOUNT - Special Register #236

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>236</td><td>ICOUNT</td><td colspan="3">Instruction count register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Debug Option</td><td>1</td><td>2 or 32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">ICOUNT ← AR[t]
Write when CINTLEVEL ≥ ICOUNTLEVEL</td><td colspan="3">AR[t] ← ICOUNT
Defined only when CINTLEVEL ≥ ICOUNTLEVEL</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">Increment on appropriate cycles</td><td colspan="3">Debug exception</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR ICOUNT ⇒ ESYNC ⇒ RSR/XSR ICOUNT</td></tr>
<tr><td colspan="6">WSR/XSR ICOUNT⇒ ISYNC ⇒ Ending CINTLEVEL ≥ ICOUNTLEVEL</td></tr>
</table>

Table 5–174. ICOUNTLEVEL - Special Register #237

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>237</td><td>ICOUNTLEVEL</td><td colspan="3">Instruction count level register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Debug Option</td><td>1</td><td>4</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">ICOUNTLEVEL ← $AR[t]_{3..0}$
Undefined if $AR[t]_{31..4} \neq 0^{28}$
Write when CINTLEVEL ≥ old ICOUNTLEVEL
Write when CINTLEVEL ≥ new ICOUNTLEVEL</td><td colspan="3">AR[t] ← 0^{28}||ICOUNTLEVEL</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">Debug exception</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR ICOUNTLEVEL ⇒ ISYNC ⇒ Ending CINTLEVEL ≥ old ICOUNTLEVEL</td></tr>
<tr><td colspan="6">WSR/XSR ICOUNTLEVEL ⇒ ISYNC ⇒ Ending CINTLEVEL ≥ new ICOUNTLEVEL</td></tr>
</table>

Table 5–175. CCOUNT - Special Register #234

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>234</td><td>CCOUNT</td><td colspan="3">Cycle count register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Timer Interrupt Option</td><td>1</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">CCOUNT ← AR[t]
Precise cycle of write is not defined
Not usually written during normal operation.</td><td colspan="3">AR[t] ← CCOUNT
Precise cycle of read is not defined.</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">Increment each cycle</td><td colspan="3">Generates Timer Interrupt</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR CCOUNT⇒ ESYNC ⇒ RSR/XSR CCOUNT</td></tr>
</table>

Table 5–176. **CCOMPARE0..2 – Special Register #240-242**

SR#	Name	Description		Reset Value
240-242	CCOMPARE0..2	Cycle count compare registers		Undefined
Option	**Count**	**Bits**	**Privileged?**	**XSR Legal?**
Timer Interrupt Option	NCCOMPARE	32	Yes	Yes
WSR Function		**RSR Function**		
CCOMPARE[sr$_{1..0}$] ← AR[t] INTERRUPT$_i$ ← 0; i is position of timer interrupt		AR[t] ← CCOMPARE[sr$_{1..0}$] AR[t] is undefined if sr$_{1..0}$ ≥ NCOMPARE		
Other Changes to the Register		**Other Effects of the Register**		
		Timer Interrupt		
Instruction ⇒ xSYNC ⇒ Instruction				
WSR/XSR CCOMPARE0..2 ⇒ ESYNC ⇒ RSR/XSR CCOUNT (to ensure CCOUNT<CCOMPAREn) WSR/XSR CCOMPARE0..2⇒ RSYNC ⇒ Any instruction which must execute after the update				

5.3.12 Breakpoint Special Registers

The Special Registers that manage the handling of breakpoint exceptions are described in Table 5–177 through Table 5–180.

Table 5–177. **IBREAKENABLE – Special Register #96**

SR#	Name	Description		Reset Value
96	IBREAKENABLE	Instruction breakpoint enable register		$0^{NIBREAK}$
Option	**Count**	**Bits**	**Privileged?**	**XSR Legal?**
Debug Option	1	NIBREAK	Yes	Yes
WSR Function		**RSR Function**		
IBREAKENABLE ← AR[t]$_{NIBREAK-1..0}$ Undefined if AR[t]$_{31..NIBREAK}$ ≠ 0^{32-NIB}		AR[t] ← $0^{32-NIBREAK}\Vert$IBREAKENABLE		
Other Changes to the Register		**Other Effects of the Register**		
		Any instruction fetch		
Instruction ⇒ xSYNC ⇒ Instruction				
WSR/XSR IBREAKENABLE ⇒ ISYNC ⇒ Any instruction access that might raise a breakpoint				

Table 5–178. IBREAKA0..1 – Special Register #128-129

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>128-129</td><td>IBREAKA0..1</td><td colspan="3">Instruction breakpoint address registers</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Debug Option</td><td>NIBREAK</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">IBREAKA[$sr_{3..0}$] ← AR[t]
Operation is undefined if $sr_{3..0}$ ≥ NIBREAK</td><td colspan="3">AR[t] ← IBREAKA[$sr_{3..0}$]
AR[t] is undefined if $sr_{3..0}$ ≥ NIBREAK</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">Any instruction fetch</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR IBREAKA0..1 ⇒ ISYNC ⇒ Any instruction access which might raise that breakpoint</td></tr>
</table>

Table 5–179. DBREAKC0..1 – Special Register #160-161

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>160-161</td><td>DBREAKC0..1</td><td colspan="3">Data breakpoint control registers</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Debug Option</td><td>NDBREAK</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">DBREAKC[$sr_{3..0}$] ← AR[t]
Operation is undefined if $sr_{3..0}$ ≥ NDBREAK</td><td colspan="3">AR[t] ← DBREAKC[$sr_{3..0}$]
AR[t] is undefined if $sr_{3..0}$ ≥ NDBREAK</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">Any data access</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR DBREAKC0..1 ⇒ DSYNC ⇒ Any load/store access which might raise that breakpoint</td></tr>
</table>

Table 5–180. DBREAKA0..1 – Special Register #144-145

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>144-145</td><td>DBREAKA0..1</td><td colspan="3">Data breakpoint address registers</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Debug Option</td><td>NDBREAK</td><td>32</td><td>Yes</td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">DBREAKA[$\mathtt{sr}_{3..0}$] ← AR[t]
Operation is undefined if $\mathtt{sr}_{3..0}$ ≥ NDBREAK</td><td colspan="3">AR[t] ← DBREAKA[$\mathtt{sr}_{3..0}$]
AR[t] is undefined if $\mathtt{sr}_{3..0}$ ≥ NDBREAK</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">Any data access</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR DBREAKA0..1 ⇒ DSYNC ⇒ Any load/store access which might raise that breakpoint</td></tr>
</table>

5.3.13 Other Privileged Special Registers

The Special Registers for other purposes are described in Table 5–181 through Table 5–186.

Table 5–181. PRID – Special Register #235

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>235</td><td>PRID</td><td colspan="3">Processor identification register</td><td>Pins</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Processor ID Option</td><td>1</td><td>32</td><td>Yes</td><td>No</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">Reserved</td><td colspan="3">AR[t] ← PRID</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">Trailing edge of RESET</td><td colspan="3"></td></tr>
</table>

Table 5–182. MMID – Special Register #89

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>89</td><td>MMID</td><td colspan="3">Memory map identification register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Trace Port Option</td><td>1</td><td>32</td><td>Yes</td><td>No</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">ID written to Trace Port</td><td colspan="3">Reserved</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3"></td></tr>
</table>

Table 5–183. DDR – Special Register #104

SR#	**Name**	**Description**			**Reset Value**
104	DDR	Debug data register			Undefined
Option		**Count**	**Bits**	**Privileged?**	**XSR Legal?**
Debug Option[1]		1	32	Yes	Yes
WSR Function			**RSR Function**		
DDR ← AR[t][2]			AR[t] ← DDR[2]		
Other Changes to the Register			**Other Effects of the Register**		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR DDR⇒ ESYNC ⇒ RSR/XSR DDR					

[1] The DDR register is actually created by the OCD Option but is listed with the Debug Option, which is a prerequisite for the OCD Option.

[2] In some implementations the DDR state is different for reads and writes; WSR.DDR followed by RSR.DDR may not return the original value.

Table 5–184. CPENABLE – Special Register #224

SR#	**Name**	**Description**			**Reset Value**
224	CPENABLE	Coprocessor enable register			Undefined
Option		**Count**	**Bits**	**Privileged?**	**XSR Legal?**
Coprocessor Option		1	1-8	Yes	Yes
WSR Function			**RSR Function**		
CPENABLE ← $\mathtt{AR[t]}_{7..0}$ Undefined if $\mathtt{AR[t]}_{31..8} \neq 0^{24}$			AR[t] ← 0^{24}‖CPENABLE (Bits corresponding to unused coprocessors are not defined on read.)		
Other Changes to the Register			**Other Effects of the Register**		
			Every coprocessor instruction		

Table 5–185. MISC0..3 – Special Register #244-247

SR#	**Name**	**Description**			**Reset Value**
244-247	MISC0..3	Miscellaneous special registers			Undefined
Option		**Count**	**Bits**	**Privileged?**	**XSR Legal?**
Miscellaneous Special Registers Option		NMISC	32	Yes	Yes
WSR Function			**RSR Function**		
$\mathtt{MISC[sr}_{1..0}\mathtt{]}$ ← AR[t]			AR[t] ← $\mathtt{MISC[sr}_{1..0}\mathtt{]}$ AR[t] is undefined if $\mathtt{sr}_{1..0} \geq \mathtt{NMISC}$		
Other Changes to the Register			**Other Effects of the Register**		

Table 5–186. **ATOMCTL** – Special Register #99

SR#	**Name**	**Description**		**Reset Value**
99	ATOMCTL	Atomic Operation Control		0x28
Option	**Count**	**Bits**	**Privileged?**	**XSR Legal?**
Conditional Store Option	1	6	Yes	Yes
WSR Function		**RSR Function**		
ATOMCTL ← AR[t]		AR[t] ← ATOMCTL		
Other Changes to the Register		**Other Effects of the Register**		
		Function of S32C1I		

5.4 User Registers

User Registers hold state added in support of designer's TIE and in some cases of options that Tensilica provides. See the *Tensilica Instruction Extension (TIE) Language User's Guide* for more information on adding User Registers to a design. Table 5–187 shows the User Registers in numerical order with references to a more detailed description. User Registers with numbers greater than or equal to 224 but not listed in Table 5–187 are reserved for future use.

Table 5–187. Numerical List of User Registers

Name[1]	Description	Required Configuration Option	User Register Number	More Detail
	Available for designer extensions		0-223	
THREADPTR	Thread pointer	Thread Pointer Option	231	Table 5–188
FCR	Floating point control register	Floating-Point Coprocessor Option	232	Table 5–189
FSR	Floating point status register	Floating-Point Coprocessor Option	233	Table 5–190

[1] Used in RUR and WUR instructions.

5.4.1 Reading and Writing User Registers

Use the RUR.* and WUR.* instructions to access the user registers. The accesses to the User Registers act as separate instructions in many ways. Replace the '*' in the instructions with the name of the User Register as specified by the designer or given in Table 5–189 and Table 5–190.

RUR.* instructions move values from a User Register to a general (AR) register. WUR.* instructions move values from a general (AR) register to a User Register. The User Registers are fully interlocked in hardware and do not need SYNC instructions.

5.4.2 The List of User Registers

Table 5–188 throughTable 5–190 list detailed information for each of the User Registers that Tensilica Options define.

The first row shows the User Register number, the name (which is used in the RUR.*, WUR.* instruction names), a short description, and the value immediately after reset.

The second row shows the Option that creates the User Register, the count or number of such User Registers, the number of bits in the User Register, and whether access to the register is privileged (requires CRING=0) or not. The option that creates the User Register is described in Chapter 4 including more information on each User Register.

The third row shows the function of the WUR.* and RUR.* instructions for this User Register.

The fourth row shows the other instructions that affect or are affected by this User Register.

The last row of each User Register's table shows that SYNC instructions are not required.

User Registers 0-223 are reserved for designer's use, and are never used by Tensilica Options. User Registers 224-255 can be used by a designer but their use may prohibit compatibility with some Tensilica-provided Options either now or in the future. Additional state registers may be added without built-in access instructions.

Table 5–188. THREADPTR – User Register #231

<table>
<tr><th>UR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>231</td><td>THREADPTR</td><td colspan="3">Thread pointer</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th></th></tr>
<tr><td colspan="2">Thread Pointer Option</td><td>1</td><td>32</td><td>No</td><td></td></tr>
<tr><th colspan="3">WUR Function</th><th colspan="3">RUR Function</th></tr>
<tr><td colspan="3">THREADPTR ← AR[t]</td><td colspan="3">AR[t] ← THREADPTR</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3"></td></tr>
</table>

Table 5–189. `FCR` – User Register #232

<table>
<tr><th>UR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>232</td><td>FCR</td><td colspan="3">Floating point control register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th></th></tr>
<tr><td colspan="2">Floating-Point Coprocessor Option</td><td>1</td><td>7</td><td>No</td><td></td></tr>
<tr><th colspan="3">WUR Function</th><th colspan="3">RUR Function</th></tr>
<tr><td colspan="3">FCR ← AR[t]</td><td colspan="3">AR[t] ← FCR</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">Most floating point computations</td></tr>
</table>

Table 5–190. `FSR` – User Register #233

<table>
<tr><th>UR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>233</td><td>FSR</td><td colspan="3">Floating point status register</td><td>Undefined</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th></th></tr>
<tr><td colspan="2">Floating-Point Coprocessor Option</td><td>1</td><td>5</td><td>No</td><td></td></tr>
<tr><th colspan="3">WUR Function</th><th colspan="3">RUR Function</th></tr>
<tr><td colspan="3">FSR ← AR[t]</td><td colspan="3">AR[t] ← FSR</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3">Most floating point computations</td><td colspan="3"></td></tr>
</table>

5.5 TLB Entries

Although some information for the instruction and data TLBs is held in the Special Registers, the protection and translation entries themselves are held in a special type of state called ITLB entries and DTLB entries. These entries are added by the Region Protection Option and the MMU Option.

These entries are accessed by special instructions for reading and writing the entries. There are also instructions for probing to see if an entry exists that will match a particular virtual address. In addition, there are instructions for invalidating particular entries. The instructions added for these purposes are listed under the Region Protection Option and the MMU Option.

After changing an Instruction TLB entry, an `ISYNC` must be executed before executing any instruction that is accessed using that TLB. After changing a data TLB entry, a `DSYNC` must be executed before any load or store that uses that entry (see Section 4.6.3.3, Section 4.6.4.2, Section 4.6.5.5, and Section 4.6.5.8 for more detailed information).

5.6 Additional Register Files

Additional register files also hold state added in support of designer's TIE and in some cases of Tensilica-provided Options. There are no built-in instructions for accessing added register files in the same manner as the `RUR.*`, and `WUR.*` instructions can be used to access the user registers. See the *Tensilica Instruction Extension (TIE) Language User's Guide* for more information on adding register files to a design.

As shown in Table 5–127, the Floating-Point Coprocessor Option creates the `FR` register file, which is an instance of this capability in a Tensilica-provided Option. The `FR` register file contains sixteen registers of 32 bits each in support of the floating point instruction set. There is no windowing in the `FR` register file.

Reads from and writes to these additional register files are always interlocked by hardware. No synchronization instructions are ever required by them.

The contents of these additional register files are undefined after reset.

5.7 Caches and Local Memories

Local memories are always architectural state. However, for many purposes caches are not architectural state in that they merely reflect the contents of main memory but provide lower latency access for the processor. When considering the cache control instructions added with the caches or the requirements placed upon software for maintaining coherence between processors/devices in their views of memory, caches sometimes act like architectural state.

Section 4.5.2 through Section 4.5.12 describe the options for adding caches and local memories to Xtensa processors.

Self-modifying code is not automatically supported in Xtensa processors. The instruction cache is not kept coherent with main memory because there is no hardware for observing writes to memory and determining whether or not those writes could have any affect on the instruction cache. Any time memory that could possibly be contained in the instruction cache is changed, the OS must ensure that the changes have been written back to system memory and invalidate either the specific locations that have been changed or else the entire instruction cache. See the description of the `ISYNC` instruction for more details.

In addition, because the instruction unit of the Xtensa processor fetches ahead, synchronization instructions are needed whenever an instruction local memory or instruction cache is modified before it can be certain that the instruction fetch engine will see the changes. For local memories, this means an `ISYNC` instruction is needed after any change to the instruction memory and before the execution of any instruction involved in the change. For instruction caches, this means an `ISYNC` instruction is needed after any change to the cache data, or the cache tag (including the invalidation required when main memory that could possibly be held in the icache is modified) and before the execution of any instruction involved in the change.

The operation of all instructions to data local memory or data cache is fully interlocked in hardware. And except for the instruction fetch discussed above, the operation of all instructions to instruction local memory or instruction cache is fully interlocked in hardware. Loads and stores, tag accesses, cache invalidations, cache line locks/unlocks, prefetches, and write backs all operate in order to the same cache locations because of the hardware interlocking. Accesses to different addresses are not necessarily in order (see Section 4.3.12.1).

Both the data and the tag stores of instruction caches and data caches are ordinary synchronous SRAMs, which are not expected to be defined after reset.

6. Instruction Descriptions

This chapter describes, in alphabetical order, each of the Xtensa ISA instructions in the Core Architecture described in Chapter 3, or in Architecture Options described in Chapter 4.

Before reading this chapter, Tensilica recommends reviewing the notation defined in Table 2–6 on page 21, Uses Of Instruction Fields.

Note that instructions with a “Required Configuration Option” specification other than “Core Architecture” are illegal if the corresponding option is not enabled, and will raise an illegal instruction exception.

The instruction word included with each instruction is the little-endian version (see Section 2.1 “Bit and Byte Order” and Chapter 7 "Instruction Formats and Opcodes" on page 569). The big-endian instruction word may be determined for any instruction by separating the little-endian instruction word at the vertical bars and reassembling the pieces in the reverse order. For example, following is the little-endian instruction word shown on page 273 for the `BEQI` instruction:

23 16	15 12	11 8	7 6	5 4	3 0
imm8	r	s	0 0	1 0	0 1 1 0
8	4	4	2	2	4

Following is the derived big-endian instruction word for the `BEQI` instruction:

0 3	4 5	6 7	8 11	12 15	16 23
0 1 1 0	1 0	0 0	s	r	imm8
4	2	2	4	4	8

The format listed after the instruction word at the top of each instruction page can also be used along with Section 7.1 “Formats” to derive the big-endian encoding.

For each instruction, the exceptions that can possibly result from its execution are listed. Because many of the potential exceptions are common to a large number of instructions, exception groups are used to save space and improve understanding. Following are the common exception groups that are referenced in the instructions. A reference to one of these groups means that any of the exceptions in the group can be raised by that instruction. Note that the groups often include previous groups.

In the following groups and in the instruction descriptions, GenExcep() is a general exception that goes to `UserExceptionVector`, `KernelExceptionVector`, or `DoubleExceptionVector`; the parentheses contain the cause that will appear in `EXCCAUSE`. DebugExcep() is a debug exception that goes to the high level interrupt for debug and the parentheses contain the cause that will appear in `DEBUGCAUSE`. WindowOverExcep is one of the three sizes of windowed register overflow exceptions[1] and WindowUnderExcep is one of the three sizes of windowed register underflow exceptions[2]. After any exceptions in the list there is an option without which that exception cannot be taken.

EveryInst Group:

- GenExcep(InstructionFetchErrorCause) if Exception Option
- GenExcep(InstTLBMissCause) if Region Protection Option or MMU Option
- GenExcep(InstTLBMultiHitCause) if Region Protection Option or MMU Option
- GenExcep(InstFetchPrivilegeCause) if Region Protection Option or MMU Option
- GenExcep(InstFetchProhibitedCause) if Region Protection Option or MMU Option
- MemoryErrorException on Instruction-fetch if Memory ECC/Parity Option
- DebugExcep(ICOUNT) if Debug Option
- DebugExcep(IBREAK) if Debug Option

EveryInstR Group:

- EveryInst Group (see page 244)
- WindowOverExcep if Windowed Register Option

Memory Group:

- EveryInstR Group (see page 244)
- GenExcep(LoadStoreErrorCause) if Exception Option
- GenExcep(LoadStoreTLBMissCause) if Region Protection Option or MMU Option
- GenExcep(LoadStoreTLBMultiHitCause) if Region Protection Option or MMU Option
- GenExcep(LoadStorePrivilegeCause) if Region Protection Option or MMU Option
- MemoryErrorException on non-Instruction-fetch if Memory ECC/Parity Option

Memory Load Group:

- Memory Group (see page 244)

[1] `WindowOverflow4`, `WindowOverflow8`, or `WindowOverflow12`.

[2] `WindowUnderflow4`, `WindowUnderflow8`, or `WindowUnderflow12`.

- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option
- GenExcep(LoadStoreAlignmentCause) if Unaligned Exception Option
- DebugExcep(DBREAK) if Debug Option

Memory Store Group:

- Memory Group (see page 244)
- GenExcep(StoreProhibitedCause) if Region Protection Option or MMU Option
- GenExcep(LoadStoreAlignmentCause) if Unaligned Exception Option
- DebugExcep(DBREAK) if Debug Option

ABS Absolute Value

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 0	0 0 0 0	r	0 0 0 1	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
ABS ar, at
```

Description

`ABS` calculates the absolute value of the contents of address register `at` and writes it to address register `ar`. Arithmetic overflow is not detected.

Operation

```
AR[r] ← if $AR[t]_{31}$ then −AR[t] else AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

Absolute Value Single ABS.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	1 0 1 0	r	s	0 0 0 1	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
ABS.S fr, fs
```

Description

`ABS.S` computes the single-precision absolute value of the contents of floating-point register `fs` and writes the result to floating-point register `fr`.

Operation

```
FR[r] ← $abs_S$(FR[s])
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

ADD Add

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 0	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
ADD ar, as, at
```

Description

`ADD` calculates the two's complement 32-bit sum of address registers `as` and `at`. The low 32 bits of the sum are written to address register `ar`. Arithmetic overflow is not detected.

`ADD` is a 24-bit instruction. The `ADD.N` density-option instruction performs the same operation in a 16-bit encoding.

Assembler Note

The assembler may convert `ADD` instructions to `ADD.N` when the Code Density Option is enabled. Prefixing the `ADD` instruction with an underscore (`_ADD`) disables this optimization and forces the assembler to generate the wide form of the instruction.

Operation

```
AR[r] ← AR[s] + AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

Narrow Add ADD.N

Instruction Word (RRRN)

15 12	11 8	7 4	3 0
r	s	t	1 0 1 0
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53)

Assembler Syntax

```
ADD.N ar, as, at
```

Description

This performs the same operation as the ADD instruction in a 16-bit encoding.

ADD.N calculates the two's complement 32-bit sum of address registers as and at. The low 32 bits of the sum are written to address register ar. Arithmetic overflow is not detected.

Assembler Note

The assembler may convert ADD.N instructions to ADD. Prefixing the ADD.N instruction with an underscore (_ADD.N) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

```
AR[r] ← AR[s] + AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

ADD.S Add Single

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
ADD.S fr, fs, ft
```

Description

ADD.S computes the IEEE754 single-precision sum of the contents of floating-point registers fs and ft, and writes the result to floating-point register fr.

Operation

```
FR[r] ← FR[s] $+_s$ FR[t]
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Add Immediate ADDI

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 1 0 0	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
ADDI at, as, -128..127
```

Description

ADDI calculates the two's complement 32-bit sum of address register as and a constant encoded in the imm8 field. The low 32 bits of the sum are written to address register at. Arithmetic overflow is not detected.

The immediate operand encoded in the instruction can range from -128 to 127. It is decoded by sign-extending imm8.

ADDI is a 24-bit instruction. The ADDI.N density-option instruction performs a similar operation (the immediate operand has less range) in a 16-bit encoding.

Assembler Note

The assembler may convert ADDI instructions to ADDI.N when the Code Density Option is enabled and the immediate operand falls within the available range. If the immediate is too large the assembler may substitute an equivalent sequence. Prefixing the ADDI instruction with an underscore (_ADDI) disables these optimizations and forces the assembler to generate the wide form of the instruction or an error instead.

Operation

```
AR[t] ← AR[s] + ($imm8_7^{24}$||imm8)
```

Exceptions

- EveryInstR Group (see page 244)

ADDI.N Narrow Add Immediate

Instruction Word (RRRN)

15 12	11 8	7 4	3 0
r	s	t	1 0 1 1
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53)

Assembler Syntax

```
ADDI.N ar, as, imm
```

Description

ADDI.N is similar to ADDI, but has a 16-bit encoding and supports a smaller range of immediate operand values encoded in the instruction word.

ADDI.N calculates the two's complement 32-bit sum of address register as and an operand encoded in the t field. The low 32 bits of the sum are written to address register ar. Arithmetic overflow is not detected.

The operand encoded in the instruction can be -1 or one to 15. If t is zero, then a value of -1 is used, otherwise the value is the zero-extension of t.

Assembler Note

The assembler may convert ADDI.N instructions to ADDI. Prefixing the ADDI.N instruction with an underscore (_ADDI.N) disables this optimization and forces the assembler to generate the narrow form of the instruction. In the assembler syntax, the number to be added to the register operand is specified. When the specified value is -1, the assembler encodes it as zero.

Operation

```
AR[r] ← AR[s] + (if t = $0^4$ then $1^{32}$ else $0^{28}$||t)
```

Exceptions

- EveryInstR Group (see page 244)

Add Immediate with Shift by 8 ADDMI

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 1 0 1	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
ADDMI at, as, -32768..32512
```

Description

`ADDMI` extends the range of constant addition. It is often used in conjunction with load and store instructions to extend the range of the base, plus offset the calculation.

`ADDMI` calculates the two's complement 32-bit sum of address register `as` and an operand encoded in the `imm8` field. The low 32 bits of the sum are written to address register `at`. Arithmetic overflow is not detected.

The operand encoded in the instruction can have values that are multiples of 256 ranging from -32768 to 32512. It is decoded by sign-extending `imm8` and shifting the result left by eight bits.

Assembler Note

In the assembler syntax, the value to be added to the register operand is specified. The assembler encodes this into the instruction by dividing by 256.

Operation

```
$AR[t] \leftarrow AR[s] + (imm8_7^{16} \| imm8 \| 0^8)$
```

Exceptions

- EveryInstR Group (see page 244)

ADDX2 Add with Shift by 1

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 1	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
ADDX2 ar, as, at
```

Description

ADDX2 calculates the two's complement 32-bit sum of address register as shifted left by one bit and address register at. The low 32 bits of the sum are written to address register ar. Arithmetic overflow is not detected.

ADDX2 is frequently used for address calculation and as part of sequences to multiply by small constants.

Operation

```
$AR[r] \leftarrow (AR[s]_{30..0} \| 0) + AR[t]$
```

Exceptions

- EveryInstR Group (see page 244)

Add with Shift by 2 ADDX4

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 0	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50r)

Assembler Syntax

```
ADDX4 ar, as, at
```

Description

ADDX4 calculates the two's complement 32-bit sum of address register as shifted left by two bits and address register at. The low 32 bits of the sum are written to address register ar. Arithmetic overflow is not detected.

ADDX4 is frequently used for address calculation and as part of sequences to multiply by small constants.

Operation

```
$AR[r] \leftarrow (AR[s]_{29..0} \| 0^{2}) + AR[t]$
```

Exceptions

- EveryInstR Group (see page 244)

ADDX8 Add with Shift by 3

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 1	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
ADDX8 ar, as, at
```

Description

ADDX8 calculates the two's complement 32-bit sum of address register as shifted left by 3 bits and address register at. The low 32 bits of the sum are written to address register ar. Arithmetic overflow is not detected.

ADDX8 is frequently used for address calculation and as part of sequences to multiply by small constants.

Operation

```
AR[r] ← $(AR[s]_{28..0} \| 0^3)$ + AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

All 4 Booleans True ALL4

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	1 0 0 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
ALL4 bt, bs
```

Description

`ALL4` sets Boolean register `bt` to the logical and of the four Boolean registers `bs+0`, `bs+1`, `bs+2`, and `bs+3`. `bs` must be a multiple of four (`b0`, `b4`, `b8`, or `b12`); otherwise the operation of this instruction is not defined. `ALL4` reduces four test results such that the result is true if all four tests are true.

When the sense of the `bs` Booleans is inverted (0 → true, 1 → false), use `ANY4` and an inverted test of the result.

Operation

```
$BR_t \leftarrow BR_{s+3}$ and $BR_{s+2}$ and $BR_{s+1}$ and $BR_{s+0}$
```

Exceptions

- EveryInst Group (see page 244)

ALL8 All 8 Booleans True

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	1 0 1 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
ALL8 bt, bs
```

Description

`ALL8` sets Boolean register `bt` to the logical and of the eight Boolean registers `bs+0`, `bs+1`, … `bs+6`, and `bs+7`. `bs` must be a multiple of eight (`b0` or `b8`); otherwise the operation of this instruction is not defined. `ALL8` reduces eight test results such that the result is true if all eight tests are true.

When the sense of the `bs` Booleans is inverted (0 → true, 1 → false), use `ANY8` and an inverted test of the result.

Operation

```
$BR_t \leftarrow BR_{s+7}$ and ... and $BR_{s+0}$
```

Exceptions

- EveryInst Group (see page 244)

Bitwise Logical And AND

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 1	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
AND ar, as, at
```

Description

`AND` calculates the bitwise logical and of address registers `as` and `at`. The result is written to address register `ar`.

Operation

```
AR[r] ← AR[s] and AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

ANDB Boolean And

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
ANDB br, bs, bt
```

Description

ANDB performs the logical and of Boolean registers bs and bt and writes the result to Boolean register br.

When the sense of one of the source Booleans is inverted (0 → true, 1 → false), use ANDBC. When the sense of both of the source Booleans is inverted, use ORB and an inverted test of the result.

Operation

```
$BR_r \leftarrow BR_s$ and $BR_t$
```

Exceptions

- EveryInst Group (see page 244)

Boolean And with Complement ANDBC

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 1	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
ANDBC br, bs, bt
```

Description

`ANDBC` performs the logical and of Boolean register `bs` with the logical complement of Boolean register `bt`, and writes the result to Boolean register `br`.

Operation

```
$BR_r \leftarrow BR_s$ and not $BR_t$
```

Exceptions

- EveryInst Group (see page 244)

ANY4 Any 4 Booleans True

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	1 0 0 0	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
ANY4 bt, bs
```

Description

ANY4 sets Boolean register bt to the logical or of the four Boolean registers bs+0, bs+1, bs+2, and bs+3. bs must be a multiple of four (b0, b4, b8, or b12); otherwise the operation of this instruction is not defined. ANY4 reduces four test results such that the result is true if any of the four tests are true.

When the sense of the bs Booleans is inverted (0 → true, 1 → false), use ALL4 and an inverted test of the result.

Operation

$$BR_t \leftarrow BR_{s+3} \text{ or } BR_{s+2} \text{ or } BR_{s+1} \text{ or } BR_{s+0}$$

Exceptions

- EveryInst Group (see page 244)

Any 8 Booleans True ANY8

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	1 0 1 0	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
ANY8 bt, bs
```

Description

ANY8 sets Boolean register bt to the logical or of the eight Boolean registers bs+0, bs+1, … bs+6, and bs+7. bs must be a multiple of eight (b0 or b8); otherwise the operation of this instruction is not defined. ANY8 reduces eight test results such that the result is true if any of the eight tests are true.

When the sense of the bs Booleans is inverted (0 → true, 1 → false), use ALL8 and an inverted test of the result.

Operation

```
$BR_t \leftarrow BR_{s+7}$ or ... or $BR_{s+0}$
```

Exceptions

- EveryInst Group (see page 244)

BALL — Branch if All Bits Set

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 0 0	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BALL as, at, label
```

Description

BALL branches if all the bits specified by the mask in address register at are set in address register as. The test is performed by taking the bitwise logical and of at and the complement of as, and testing if the result is zero.

The target instruction address of the branch is given by the address of the BALL instruction, plus the sign-extended 8-bit imm8 field of the instruction plus four. If any of the masked bits are clear, execution continues with the next sequential instruction.

The inverse of BALL is BNALL.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BALL) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if ((not AR[s]) and AR[t]) = $0^{32}$ then
     nextPC ← PC + ($\text{imm8}_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 0 0 0	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BANY as, at, label
```

Description

`BANY` branches if any of the bits specified by the mask in address register `at` are set in address register `as`. The test is performed by taking the bitwise logical and of `as` and `at` and testing if the result is non-zero.

The target instruction address of the branch is given by the address of the `BANY` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If all of the masked bits are clear, execution continues with the next sequential instruction.

The inverse of `BANY` is `BNONE`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BANY`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if (AR[s] and AR[t]) $\neq 0^{32}$ then
     nextPC $\leftarrow$ PC + ($\text{imm8}_7^{24} \| \text{imm8}$) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BBC Branch if Bit Clear

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 0 1	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BBC as, at, label
```

Description

`BBC` branches if the bit specified by the low five bits of address register `at` is clear in address register `as`. For little-endian processors, bit 0 is the least significant bit and bit 31 is the most significant bit. For big-endian processors, bit 0 is the most significant bit and bit 31 is the least significant bit.

The target instruction address of the branch is given by the address of the `BBC` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the specified bit is set, execution continues with the next sequential instruction.

The inverse of `BBC` is `BBS`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BBC`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
$b$ ← AR[t]$_{4..0}$ xor msbFirst$^{5}$
if AR[s]$_{b}$ = 0 then
     nextPC ← PC + (imm8$_{7}^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Branch if Bit Clear Immediate BBCI

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 bbi_4	s	$bbi_{3..0}$	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BBCI as, 0..31, label
```

Description

`BBCI` branches if the bit specified by the constant encoded in the `bbi` field of the instruction word is clear in address register `as`. For little-endian processors, bit 0 is the least significant bit and bit 31 is the most significant bit. For big-endian processors bit 0 is the most significant bit and bit 31 is the least significant bit. The `bbi` field is split, with bits 3..0 in bits 7..4 of the instruction word, and bit 4 in bit 12 of the instruction word.

The target instruction address of the branch is given by the address of the `BBCI` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the specified bit is set, execution continues with the next sequential instruction.

The inverse of `BBCI` is `BBSI`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BBCI`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
$b$ ← bbi xor msbFirst$^5$
if AR[s]$_b$ = 0 then
     nextPC ← PC + (imm8$_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BBCI.L Branch if Bit Clear Immediate LE

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 bbi_4	s	$bbi_{3..0}$	0 1 1 1
8	4	4	4	4

Required Configuration Option

Assembler Macro

Assembler Syntax

```
BBCI.L as, 0..31, label
```

Description

`BBCI.L` is an assembler macro for `BBCI` that always uses little-endian bit numbering. That is, it branches if the bit specified by its immediate is clear in address register `as`, where bit 0 is the least significant bit and bit 31 is the most significant bit.

The inverse of `BBCI.L` is `BBSI.L`.

Assembler Note

For little-endian processors, `BBCI.L` and `BBCI` are identical. For big-endian processors, the assembler will convert `BBCI.L` instructions to `BBCI` by changing the encoded immediate value to `31-imm`.

Exceptions

- EveryInstR Group (see page 244)

Branch if Bit Set BBS

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 1 0 1	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BBS as, at, label
```

Description

BBS branches if the bit specified by the low five bits of address register at is set in address register as. For little-endian processors, bit 0 is the least significant bit and bit 31 is the most significant bit. For big-endian processors, bit 0 is the most significant bit and bit 31 is the least significant bit.

The target instruction address of the branch is given by the address of the BBS instruction, plus the sign-extended 8-bit imm8 field of the instruction plus four. If the specified bit is clear, execution continues with the next sequential instruction.

The inverse of BBS is BBC.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BBS) disables this feature and forces the assembler to generate an error in this case.

Operation

```
b ← AR[t]$_{4..0}$ xor msbFirst$^{5}$
if AR[s]$_{b}$ ≠ 0 then
    nextPC ← PC + (imm8$_{7}^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BBSI Branch if Bit Set Immediate

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 1 1 bbi_4	s	$\text{bbi}_{3..0}$	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BBSI as, 0..31, label
```

Description

BBSI branches if the bit specified by the constant encoded in the bbi field of the instruction word is set in address register as. For little-endian processors, bit 0 is the least significant bit and bit 31 is the most significant bit. For big-endian processors, bit 0 is the most significant bit and bit 31 is the least significant bit. The bbi field is split, with bits 3..0 in bits 7..4 of the instruction word, and bit 4 in bit 12 of the instruction word.

The target instruction address of the branch is given by the address of the BBSI instruction, plus the sign-extended 8-bit imm8 field of the instruction plus four. If the specified bit is clear, execution continues with the next sequential instruction.

The inverse of BBSI is BBCI.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BBSI) disables this feature and forces the assembler to generate an error in this case.

Operation

```
$b$ ← bbi xor $\text{msbFirst}^5$
if $\text{AR[s]}_b \neq 0$ then
    nextPC ← PC + $(\text{imm8}_7^{24} \| \text{imm8}) + 4$
endif
```

Exceptions

- EveryInstR Group (see page 244)

Branch if Bit Set Immediate LE BBSI.L

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 1 1 bbi_4	s	bbi	0 1 1 1
8	4	4	4	4

Required Configuration Option

Assembler Macro

Assembler Syntax

```
BBSI.L as, 0..31, label
```

Description

`BBSI.L` is an assembler macro for `BBSI` that always uses little-endian bit numbering. That is, it branches if the bit specified by its immediate is set in address register `as`, where bit 0 is the least significant bit and bit 31 is the most significant bit.

The inverse of `BBSI.L` is `BBCI.L`.

Assembler Note

For little-endian processors, `BBSI.L` and `BBSI` are identical. For big-endian processors, the assembler will convert `BBSI.L` instructions to `BBSI` by changing the encoded immediate value to `31-imm`.

Exceptions

- EveryInstR Group (see page 244)

BEQ Branch if Equal

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 0 0 1	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BEQ as, at, label
```

Description

`BEQ` branches if address registers `as` and `at` are equal.

The target instruction address of the branch is given by the address of the `BEQ` instruction plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the registers are not equal, execution continues with the next sequential instruction.

The inverse of `BEQ` is `BNE`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BEQ`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if AR[s] = AR[t] then
    nextPC ← PC + ($imm8_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Branch if Equal Immediate BEQI

Instruction Word (RRI8)

23 16	15 12	11 8	7	6	5	4	3			0
imm8	r	s	0	0	1	0	0	1	1	0
8	4	4	2		2		4			

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BEQI as, imm, label
```

Description

BEQI branches if address register as and a constant encoded in the r field are equal. The constant values encoded in the r field are not simply 0..15. For the constant values that can be encoded by r, see Table 3–17 on page 41.

The target instruction address of the branch is given by the address of the BEQI instruction, plus the sign-extended 8-bit imm8 field of the instruction plus four. If the register is not equal to the constant, execution continues with the next sequential instruction.

The inverse of BEQI is BNEI.

Assembler Note

The assembler may convert BEQI instructions to BEQZ or BEQZ.N when given an immediate operand that evaluates to zero. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BEQI) disables these features and forces the assembler to generate an error instead.

Operation

```
if AR[s] = B4CONST(r) then
     nextPC ← PC + ($imm8_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BEQZ Branch if Equal to Zero

Instruction Word BRI12

23	12	11	8	7	6	5	4	3			0
imm12		s		0	0	0	1	0	1	1	0
12		4		2		2		4			

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BEQZ as, label
```

Description

BEQZ branches if address register as is equal to zero. BEQZ provides 12 bits of target range instead of the eight bits available in most conditional branches.

The target instruction address of the branch is given by the address of the BEQZ instruction, plus the sign-extended 12-bit imm12 field of the instruction plus four. If register as is not equal to zero, execution continues with the next sequential instruction.

The inverse of BEQZ is BNEZ.

Assembler Note

The assembler may convert BEQZ instructions to BEQZ.N when the Code Density Option is enabled and the branch target is reachable with the shorter instruction. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BEQZ) disables these features and forces the assembler to generate the wide form of the instruction and an error when the label is out of range).

Operation

```
if AR[s] = $0^{32}$ then
    nextPC ← PC + ($\text{imm12}_{11}{}^{20}$||imm12) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Narrow Branch if Equal Zero BEQZ.N

Instruction Word (RI6)

15 12	11 8	7 4	3 0
imm6$_{3..0}$	s	1 0 imm6$_{5..4}$	1 1 0 0
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53)

Assembler Syntax

```
BEQZ.N as, label
```

Description

This performs the same operation as the `BEQZ` instruction in a 16-bit encoding. `BEQZ.N` branches if address register `as` is equal to zero. `BEQZ.N` provides six bits of target range instead of the 12 bits available in `BEQZ`.

The target instruction address of the branch is given by the address of the `BEQZ.N` instruction, plus the zero-extended 6-bit `imm6` field of the instruction plus four. Because the offset is unsigned, this instruction can only be used to branch forward. If register `as` is not equal to zero, execution continues with the next sequential instruction.

The inverse of `BEQZ.N` is `BNEZ.N`.

Assembler Note

The assembler may convert `BEQZ.N` instructions to `BEQZ`. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BEQZ.N`) disables these features and forces the assembler to generate the narrow form of the instruction and an error when the label is out of range.

Operation

```
if AR[s] = $0^{32}$ then
     nextPC ← PC + ($0^{26}$||imm6) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BF Branch if False

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 0 0 0	s	0 1 1 1	0 1 1 0
8	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
BF bs, label
```

Description

BF branches to the target address if Boolean register bs is false.

The target instruction address of the branch is given by the address of the BF instruction plus the sign-extended 8-bit imm8 field of the instruction plus four. If the Boolean register bs is true, execution continues with the next sequential instruction.

The inverse of BF is BT.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BF) disables this feature and forces the assembler to generate an error when the label is out of range.

Operation

```
if not $BR_s$ then
     nextPC ← PC + ($imm8_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInst Group (see page 244)

Branch if Greater Than or Equal BGE

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 0 1 0	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BGE as, at, label
```

Description

`BGE` branches if address register `as` is two's complement greater than or equal to address register `at`.

The target instruction address of the branch is given by the address of the `BGE` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the address register `as` is less than address register `at`, execution continues with the next sequential instruction.

The inverse of `BGE` is `BLT`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BGE`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if AR[s] ≥ AR[t] then
     nextPC ← PC + $(\mathrm{imm8}_7^{24} \| \mathrm{imm8})$ + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BGEI Branch if Greater Than or Equal Immediate

Instruction Word (BRI8)

23 16	15 12	11 8	7 6	5 4	3 0
imm8	r	s	1 1	1 0	0 1 1 0
8	4	4	2	2	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BGEI as, imm, label
```

Description

BGEI branches if address register as is two's complement greater than or equal to the constant encoded in the r field. The constant values encoded in the r field are not simply 0..15. For the constant values that can be encoded by r, see Table 3–17 on page 41.

The target instruction address of the branch is given by the address of the BGEI instruction, plus the sign-extended 8-bit imm8 field of the instruction plus four. If the address register as is less than the constant, execution continues with the next sequential instruction.

The inverse of BGEI is BLTI.

Assembler Note

The assembler may convert BGEI instructions to BGEZ when given an immediate operand that evaluates to zero. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BGEI) disables these features and forces the assembler to generate an error instead.

Operation

```
if AR[s] ≥ B4CONST(r) then
     nextPC ← PC + ($imm8_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Branch if Greater Than or Equal Unsigned BGEU

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 0 1 1	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BGEU as, at, label
```

Description

BGEU branches if address register as is unsigned greater than or equal to address register at.

The target instruction address of the branch is given by the address of the BGEU instruction, plus the sign-extended 8-bit imm8 field of the instruction plus four. If the address register as is unsigned less than address register at, execution continues with the next sequential instruction.

The inverse of BGEU is BLTU.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BGEU) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if (0||AR[s]) ≥ (0||AR[t]) then
     nextPC ← PC + ($imm8_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BGEUI Branch if Greater Than or Eq Unsigned Imm

Instruction Word (BRI8)

23 16	15 12	11 8	7 6	5 4	3 0
imm8	r	s	1 1	1 1	0 1 1 0
8	4	4	2	2	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BGEUI as, imm, label
```

Description

BGEUI branches if address register as is unsigned greater than or equal to the constant encoded in the r field. The constant values encoded in the r field are not simply 0..15. For the constant values that can be encoded by r, see Table 3–18 on page 42.

The target instruction address of the branch is given by the address of the BGEUI instruction plus the sign-extended 8-bit imm8 field of the instruction plus four. If the address register as is less than the constant, execution continues with the next sequential instruction.

The inverse of BGEUI is BLTUI.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BGEUI) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if (0||AR[s]) ≥ (0||B4CONSTU(r)) then
     nextPC ← PC + ($imm8_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Branch if Greater Than or Equal to Zero BGEZ

Instruction Word (BRI12)

23 12	11 8	7 6	5 4	3 0
imm12	s	1 1	0 1	0 1 1 0
12	4	2	2	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BGEZ as, label
```

Description

BGEZ branches if address register as is greater than or equal to zero (the most significant bit is clear). BGEZ provides 12 bits of target range instead of the eight bits available in most conditional branches.

The target instruction address of the branch is given by the address of the BGEZ instruction plus the sign-extended 12-bit imm12 field of the instruction plus four. If register as is less than zero, execution continues with the next sequential instruction.

The inverse of BGEZ is BLTZ.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BGEZ) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if $AR[s]_{31} = 0$ then
     $nextPC \leftarrow PC + (imm12_{11}{}^{20} \| imm12) + 4$
endif
```

Exceptions

- EveryInstR Group (see page 244)

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 0 1 0	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BLT as, at, label
```

Description

`BLT` branches if address register `as` is two's complement less than address register `at`.

The target instruction address of the branch is given by the address of the `BLT` instruction plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the address register `as` is greater than or equal to address register `at`, execution continues with the next sequential instruction.

The inverse of `BLT` is `BGE`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BLT`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if AR[s] < AR[t] then
     nextPC ← PC + ($imm8_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Branch if Less Than Immediate BLTI

Instruction Word (BRI8)

23 16	15 12	11 8	7 6	5 4	3 0
imm8	r	s	1 0	1 0	0 1 1 0
8	4	4	2	2	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BLTI as, imm, label
```

Description

BLTI branches if address register as is two's complement less than the constant encoded in the r field. The constant values encoded in the r field are not simply 0..15. For the constant values that can be encoded by r, see Table 3–17 on page 41.

The target instruction address of the branch is given by the address of the BLTI instruction plus the sign-extended 8-bit imm8 field of the instruction plus four. If the address register as is greater than or equal to the constant, execution continues with the next sequential instruction.

The inverse of BLTI is BGEI.

Assembler Note

The assembler may convert BLTI instructions to BLTZ when given an immediate operand that evaluates to zero. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BLTI) disables these features and forces the assembler to generate an error instead.

Operation

```
if AR[s] < B4CONST(r) then
     nextPC ← PC + ($\text{imm8}_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BLTU Branch if Less Than Unsigned

Instruction Word (RRI8)

23 … 16	15 … 12	11 … 8	7 … 4	3 … 0
imm8	0 0 1 1	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BLTU as, at, label
```

Description

BLTU branches if address register as is unsigned less than address register at.

The target instruction address of the branch is given by the address of the BLTU instruction, plus the sign-extended 8-bit imm8 field of the instruction plus four. If the address register as is greater than or equal to address register at, execution continues with the next sequential instruction.

The inverse of BLTU is BGEU.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BLTU) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if (0||AR[s]) < (0||AR[t]) then
     nextPC ← PC + ($imm8_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Branch if Less Than Unsigned Immediate BLTUI

Instruction Word (BRI8)

23 16	15 12	11 8	7	6	5	4	3			0
imm8	r	s	1	0	1	1	0	1	1	0
8	4	4	2		2		4			

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BLTUI as, imm, label
```

Description

BLTUI branches if address register as is unsigned less than the constant encoded in the r field. The constant values encoded in the r field are not simply 0..15. For the constant values that can be encoded by r, see Table 3–18 on page 42.

The target instruction address of the branch is given by the address of the BLTUI instruction, plus the sign-extended 8-bit imm8 field of the instruction plus four. If the address register as is greater than or equal to the constant, execution continues with the next sequential instruction.

The inverse of BLTUI is BGEUI.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BLTUI) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if (0||AR[s]) < (0||B4CONSTU(r)) then
     nextPC ← PC + ($imm8_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BLTZ

Branch if Less Than Zero

Instruction Word (BRI12)

23 12	11 8	7	6	5	4	3			0
imm12	s	1	0	0	1	0	1	1	0
12	4	2		2		4			

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BLTZ as, label
```

Description

BLTZ branches if address register as is less than zero (the most significant bit is set). BLTZ provides 12 bits of target range instead of the eight bits available in most conditional branches.

The target instruction address of the branch is given by the address of the BLTZ instruction, plus the sign-extended 12-bit imm12 field of the instruction plus four. If register as is greater than or equal to zero, execution continues with the next sequential instruction.

The inverse of BLTZ is BGEZ.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BLTZ) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if AR[s]$_{31}$ ≠ 0 then
     nextPC ← PC + (imm12$_{11}^{20}$||imm12) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Branch if Not-All Bits Set BNALL

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 1 0 0	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BNALL as, at, label
```

Description

BNALL branches if any of the bits specified by the mask in address register at are clear in address register as (that is, if they are not all set). The test is performed by taking the bitwise logical and of at with the complement of as and testing if the result is non-zero.

The target instruction address of the branch is given by the address of the BNALL instruction, plus the sign-extended 8-bit imm8 field of the instruction plus four. If all of the masked bits are set, execution continues with the next sequential instruction.

The inverse of BNALL is BALL.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BNALL) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if ((not AR[s]) and AR[t]) ≠ $0^{32}$ then
     nextPC ← PC + ($imm8_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BNE Branch if Not Equal

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 0 0 1	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BNE as, at, label
```

Description

`BNE` branches if address registers `as` and `at` are not equal.

The target instruction address of the branch is given by the address of the `BNE` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the registers are equal, execution continues with the next sequential instruction.

The inverse of `BNE` is `BEQ`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BNE`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if AR[s] $\neq$ AR[t] then
     nextPC $\leftarrow$ PC + $(\text{imm8}_7^{24} \| \text{imm8})$ + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Branch if Not Equal Immediate BNEI

Instruction Word (BRI8)

23 16	15 12	11 8	7 6	5 4	3 0
imm8	r	s	0 1	1 0	0 1 1 0
8	4	4	2	2	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BNEI as, imm, label
```

Description

BNEI branches if address register as and a constant encoded in the r field are not equal. The constant values encoded in the r field are not simply 0..15. For the constant values that can be encoded by r, see Table 3–17 on page 41.

The target instruction address of the branch is given by the address of the BNEI instruction, plus the sign-extended 8-bit imm8 field of the instruction plus four. If the register is equal to the constant, execution continues with the next sequential instruction.

The inverse of BNEI is BEQI.

Assembler Note

The assembler may convert BNEI instructions to BNEZ or BNEZ.N when given an immediate operand that evaluates to zero. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BNEI) disables these features and forces the assembler to generate an error instead.

Operation

```
if AR[s] ≠ B4CONST(r) then
     nextPC ← PC + ($\text{imm8}_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BNEZ Branch if Not-Equal to Zero

Instruction Word (BRI12)

23 12	11 8	7	6	5	4	3			0
imm12	s	0	1	0	1	0	1	1	0
12	4	2		2		4			

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BNEZ as, label
```

Description

`BNEZ` branches if address register `as` is not equal to zero. `BNEZ` provides 12 bits of target range instead of the eight bits available in most conditional branches.

The target instruction address of the branch is given by the address of the `BNEZ` instruction, plus the sign-extended 12-bit `imm12` field of the instruction plus four. If register `as` is equal to zero, execution continues with the next sequential instruction.

The inverse of `BNEZ` is `BEQZ`.

Assembler Note

The assembler may convert `BNEZ` instructions to `BNEZ.N` when the Code Density Option is enabled and the branch target is reachable with the shorter instruction. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BNEZ`) disables these features and forces the assembler to generate the `BNEZ` form of the instruction and an error when the label is out of range.

Operation

```
if AR[s] $\neq 0^{32}$ then
     nextPC $\leftarrow$ PC + $(\text{imm12}_{11}{}^{20} \| \text{imm12})$ + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Narrow Branch if Not Equal Zero BNEZ.N

Instruction Word (RI6)

15 12	11 8	7 4	3 0
$imm6_{3..0}$	s	1 1 $imm6_{5..4}$	1 1 0 0
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53))

Assembler Syntax

```
BNEZ.N as, label
```

Description

This performs the same operation as the `BNEZ` instruction in a 16-bit encoding. `BNEZ.N` branches if address register `as` is not equal to zero. `BNEZ.N` provides six bits of target range instead of the 12 bits available in `BNEZ`.

The target instruction address of the branch is given by the address of the `BNEZ.N` instruction, plus the zero-extended 6-bit `imm6` field of the instruction plus four. Because the offset is unsigned, this instruction can only be used to branch forward. If register `as` is equal to zero, execution continues with the next sequential instruction.

The inverse of `BNEZ.N` is `BEQZ.N`.

Assembler Note

The assembler may convert `BNEZ.N` instructions to `BNEZ`. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BNEZ.N`) disables these features and forces the assembler to generate the narrow form of the instruction and an error when the label is out of range.

Operation

```
if AR[s] ≠ $0^{32}$ then
    nextPC ← PC + ($0^{26}$||imm6) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

BNONE Branch if No Bit Set

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 0 0 0	s	t	0 1 1 1
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
BNONE as, at, label
```

Description

BNONE branches if all of the bits specified by the mask in address register at are clear in address register as (that is, if none of them are set). The test is performed by taking the bitwise logical and of as with at and testing if the result is zero.

The target instruction address of the branch is given by the address of the BNONE instruction, plus the sign-extended 8-bit imm8 field of the instruction plus four. If any of the masked bits are set, execution continues with the next sequential instruction.

The inverse of BNONE is BANY.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (_BNONE) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if (AR[s] and AR[t]) = $0^{32}$ then
     nextPC ← PC + ($\text{imm8}_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Breakpoint BREAK

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 1 0 0	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Debug Option (See Section 4.7.6 on page 197)

Assembler Syntax

```
BREAK 0..15, 0..15
```

Description

This instruction simply raises an exception when it is executed and `PS.INTLEVEL` < `DEBUGLEVEL`. The high-priority vector for `DEBUGLEVEL` is used. The `DEBUGCAUSE` register is written as part of raising the exception to indicate that `BREAK` raised the debug exception. The address of the `BREAK` instruction is stored in `EPC[DEBUGLEVEL]`. The `s` and `t` fields of the instruction word are not used by the processor; they are available for use by the software. When `PS.INTLEVEL` ≥ `DEBUGLEVEL`, `BREAK` is a no-op.

The `BREAK` instruction typically calls a debugger when program execution reaches a certain point (a “breakpoint”). The instruction at the breakpoint is replaced with the `BREAK` instruction. To continue execution after a breakpoint is reached, the debugger must re-write the `BREAK` to the original instruction, single-step by one instruction, and then put back the `BREAK` instruction again.

Writing instructions requires special consideration. See the `ISYNC` instruction for more information.

When it is not possible to write the instruction memory (for example, for ROM code), the `IBREAKA` feature provides breakpoint capabilities (see Debug Option).

Software can also use `BREAK` to indicate an error condition that requires the programmer’s attention. The `s` and `t` fields may encode information about the situation.

`BREAK` is a 24-bit instruction. The `BREAK.N` density-option instruction performs a similar operation in a 16-bit encoding.

Assembler Note

The assembler may convert BREAK instructions to BREAK.N when the Code Density Option is enabled and the second imm is zero. Prefixing the instruction mnemonic with an underscore (_BREAK) disables this optimization and forces the assembler to generate the wide form of the instruction.

Operation

```
if PS.INTLEVEL < DEBUGLEVEL then
     EPC[DEBUGLEVEL] ← PC
     EPS[DEBUGLEVEL] ← PS
     DEBUGCAUSE ← 001000
     nextPC ← InterruptVector[DEBUGLEVEL]
     PS.EXCM ← 1
     PS.INTLEVEL ← DEBUGLEVEL
endif
```

Exceptions

- EveryInst Group (see page 244)
- DebugExcep(BREAK) if Debug Option

Narrow Breakpoint BREAK.N

Instruction Word (RRRN)

15 12	11 8	7 4	3 0
1 1 1 1	s	0 0 1 0	1 1 0 1
4	4	4	4

Required Configuration Option

Debug Option (See Section 4.7.6 on page 197) and Code Density Option (See Section 4.3.1 on page 53)

Assembler Syntax

```
BREAK.N 0..15
```

Description

BREAK.N is similar in operation to BREAK (page 293), except that it is encoded in a 16-bit format instead of 24 bits, there is only a 4-bit imm field, and a different bit is set in DEBUGCAUSE. Use this instruction to set breakpoints on 16-bit instructions.

Assembler Note

The assembler may convert BREAK.N instructions to BREAK. Prefixing the BREAK.N instruction with an underscore (_BREAK.N) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

```
if PS.INTLEVEL < DEBUGLEVEL then
     EPC[DEBUGLEVEL] ← PC
     EPS[DEBUGLEVEL] ← PS
     DEBUGCAUSE ← 010000
     nextPC ← InterruptVector[DEBUGLEVEL]
     PS.EXCM ← 1
     PS.INTLEVEL ← DEBUGLEVEL
endif
```

Exceptions

- EveryInst Group (see page 244)
- DebugExcep(BREAK.N) if Debug Option

BT Branch if True

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 0 0 1	s	0 1 1 1	0 1 1 0
8	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)s

Assembler Syntax

```
BT bs, label
```

Description

`BT` branches to the target address if Boolean register `bs` is true.

The target instruction address of the branch is given by the address of the `BT` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the Boolean register `bs` is false, execution continues with the next sequential instruction.

The inverse of `BT` is `BF`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BT`) disables this feature and forces the assembler to generate an error when the label is out of range.

Operation

```
if $BR_s$ then
     nextPC ← PC + ($\text{imm8}_7^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInst Group (see page 244)

Non-windowed Call CALL0

Instruction Word (CALL)

23 6	5 4	3 0
offset	0 0	0 1 0 1
18	2	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
CALL0 label
```

Description

CALL0 calls subroutines without using register windows. The return address is placed in a0, and the processor then branches to the target address. The return address is the address of the CALL0 instruction plus three.

The target instruction address must be 32-bit aligned. This allows CALL0 to have a larger effective range (-524284 to 524288 bytes). The target instruction address of the call is given by the address of the CALL0 instruction with the least significant two bits set to zero plus the sign-extended 18-bit offset field of the instruction shifted by two, plus four.

The RET and RET.N instructions are used to return from a subroutine called by CALL0.

See the CALLX0 instruction (page 304) for calling routines where the target address is given by the contents of a register.

To call using the register window mechanism, see the CALL4, CALL8, and CALL12 instructions.

Operation

```
AR[0] ← PC + 3
nextPC ← $(PC_{31..2} + (offset_{17}^{12} \| offset) + 1) \| 00$
```

Exceptions

- EveryInst Group (see page 244)

CALL4 Call PC-relative, Rotate Window by 4

Instruction Word (CALL)

23 6	5	4	3			0
offset	0	1	0	1	0	1
18	2		4			

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
CALL4 label
```

Description

CALL4 calls subroutines using the register windows mechanism, requesting the callee rotate the window by four registers. The CALL4 instruction does not rotate the window itself, but instead stores the window increment for later use by the ENTRY instruction. The return address and window increment are placed in the caller's a4 (the callee's a0), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the CALL4 instruction plus three). The window increment is also stored in the CALLINC field of the PS register, where it is accessed by the ENTRY instruction.

The target instruction address must be a 32-bit aligned ENTRY instruction. This allows CALL4 to have a larger effective range (–524284 to 524288 bytes). The target instruction address of the call is given by the address of the CALL4 instruction with the two least significant bits set to zero plus the sign-extended 18-bit offset field of the instruction shifted by two, plus four.

See the CALLX4 instruction for calling routines where the target address is given by the contents of a register.

Use the RETW and RETW.N instructions to return from a subroutine called by CALL4.

The window increment stored with the return address register in a4 occupies the two most significant bits of the register, and therefore those bits must be filled in by the subroutine return. The RETW and RETW.N instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

See the CALL0 instruction for calling routines using the non-windowed subroutine protocol.

The caller’s a4..a15 are the same registers as the callee’s a0..a11 after the callee executes the ENTRY instruction. You can use these registers for parameter passing. The caller’s a0..a3 are hidden by CALL4, and therefore you can use them to keep values that are live across the call.

Operation

```
WindowCheck (00, 00, 01)
PS.CALLINC ← 01
AR[0100] ← 01||(PC + 3)$_{29..0}$
nextPC ← (PC$_{31..2}$ + (offset$_{17}^{12}$||offset) + 1)||00
```

Exceptions

- EveryInstR Group (see page 244)

CALL8 Call PC-relative, Rotate Window by 8

Instruction Word (CALL)

23 6	5	4	3			0
offset	1	0	0	1	0	1
18	2		4			

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
CALL8 label
```

Description

`CALL8` calls subroutines using the register windows mechanism, requesting the callee rotate the window by eight registers. The `CALL8` instruction does not rotate the window itself, but instead stores the window increment for later use by the `ENTRY` instruction. The return address and window increment are placed in the caller's `a8` (the callee's `a0`), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the `CALL8` instruction plus three). The window increment is also stored in the `CALLINC` field of the `PS` register, where it is accessed by the `ENTRY` instruction.

The target instruction address must be a 32-bit aligned `ENTRY` instruction. This allows `CALL8` to have a larger effective range (–524284 to 524288 bytes). The target instruction address of the call is given by the address of the `CALL8` instruction with the two least significant bits set to zero, plus the sign-extended 18-bit `offset` field of the instruction shifted by two, plus four.

See the `CALLX8` instruction for calling routines where the target address is given by the contents of a register.

Use the `RETW` and `RETW.N` instructions to return from a subroutine called by `CALL8`.

The window increment stored with the return address register in `a8` occupies the two most significant bits of the register, and therefore those bits must be filled in by the subroutine return. The `RETW` and `RETW.N` instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

Call PC-relative, Rotate Window by 8 CALL8

See the `CALL0` instruction for calling routines using the non-windowed subroutine protocol.

The caller's `a8`..`a15` are the same registers as the callee's `a0`..`a7` after the callee executes the `ENTRY` instruction. You can use these registers for parameter passing. The caller's `a0`..`a7` are hidden by `CALL8`, and therefore you may use them to keep values that are live across the call.

Operation

```
WindowCheck (00, 00, 10)
PS.CALLINC ← 10
AR[1000] ← $10 \| (PC + 3)_{29..0}$
nextPC ← $(PC_{31..2} + (offset_{17}^{12} \| offset) + 1) \| 00$
```

Exceptions

- EveryInstR Group (see page 244)

CALL12 Call PC-relative, Rotate Window by 12

Instruction Word (CALL)

23 6	5	4	3			0
offset	1	1	0	1	0	1
18	2		4			

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
CALL12 label
```

Description

CALL12 calls subroutines using the register windows mechanism, requesting the callee rotate the window by 12 registers. The CALL12 instruction does not rotate the window itself, but instead stores the window increment for later use by the ENTRY instruction. The return address and window increment are placed in the caller's a12 (the callee's a0), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the CALL12 instruction plus three). The window increment is also stored in the CALLINC field of the PS register, where it is accessed by the ENTRY instruction.

The target instruction address must be a 32-bit aligned ENTRY instruction. This allows CALL12 to have a larger effective range (–524284 to 524288 bytes). The target instruction address of the call is given by the address of the CALL12 instruction with the two least significant bits set to zero, plus the sign-extended 18-bit offset field of the instruction shifted by two, plus four.

See the CALLX12 instruction for calling routines where the target address is given by the contents of a register.

The RETW and RETW.N instructions return from a subroutine called by CALL12.

The window increment stored with the return address register in a12 occupies the two most significant bits of the register, and therefore those bits must be filled in by the subroutine return. The RETW and RETW.N instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

Call PC-relative, Rotate Window by 12 CALL12

See the CALL0 instruction for calling routines using the non-windowed subroutine protocol.

The caller's a12..a15 are the same registers as the callee's a0..a3 after the callee executes the ENTRY instruction. You can use these registers for parameter passing. The caller's a0..a11 are hidden by CALL12, and therefore you may use them to keep values that are live across the call.

Operation

```
WindowCheck (00, 00, 11)
PS.CALLINC ← 11
AR[1100] ← 11||$(PC + 3)_{29..0}$
nextPC ← $(PC_{31..2} + (offset_{17}{}^{12}||offset) + 1)||00$
```

Exceptions

- EveryInstR Group (see page 244)

CALLX0 Non-windowed Call Register

Instruction Word (CALLX)

23 20	19 16	15 12	11 8	7 6	5 4	3 0
0 0 0 0	0 0 0 0	0 0 0 0	s	1 1	0 0	0 0 0 0
4	4	4	4	2	2	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
CALLX0 as
```

Description

CALLX0 calls subroutines without using register windows. The return address is placed in a0, and the processor then branches to the target address. The return address is the address of the CALLX0 instruction, plus three.

The target instruction address of the call is given by the contents of address register as.

The RET and RET.N instructions return from a subroutine called by CALLX0.

To call using the register window mechanism, see the CALLX4, CALLX8, and CALLX12 instructions.

Operation

```
nextPC ← AR[s]
AR[0] ← PC + 3
```

Exceptions

- EveryInstR Group (see page 244)

Call Register, Rotate Window by 4 CALLX4

Instruction Word (CALLX)

23 20	19 16	15 12	11 8	7 6	5 4	3 0
0 0 0 0	0 0 0 0	0 0 0 0	s	1 1	0 1	0 0 0 0
4	4	4	4	2	2	4

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
CALLX4 as
```

Description

CALLX4 calls subroutines using the register windows mechanism, requesting the callee rotate the window by four registers. The CALLX4 instruction does not rotate the window itself, but instead stores the window increment for later use by the ENTRY instruction. The return address and window increment are placed in the caller's a4 (the callee's a0), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the CALLX4 instruction plus three). The window increment is also stored in the CALLINC field of the PS register, where it is accessed by the ENTRY instruction.

The target instruction address of the call is given by the contents of address register as. The target instruction must be an ENTRY instruction.

See the CALL4 instruction for calling routines where the target address is given by a PC-relative offset in the instruction.

The RETW and RETW.N instructions return from a subroutine called by CALLX4.

The window increment stored with the return address register in a4 occupies the two most significant bits of the register, and therefore those bits must be filled in by the subroutine return. The RETW and RETW.N instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

See the CALLX0 instruction for calling routines using the non-windowed subroutine protocol.

CALLX4 Call Register, Rotate Window by 4

The caller’s a4..a15 are the same registers as the callee’s a0..a11 after the callee executes the ENTRY instruction. You can use these registers for parameter passing. The caller’s a0..a3 are hidden by CALLX4, and therefore you may use them to keep values that are live across the call.

Operation

```
WindowCheck (00, 00, 01)
PS.CALLINC ← 01
AR[01||00] ← 01||(PC + 3)$_{29..0}$
nextPC ← AR[s]
```

Exceptions

- EveryInstR Group (see page 244)

Call Register, Rotate Window by 8 CALLX8

Instruction Word (CALLX)

23 20	19 16	15 12	11 8	7 6	5 4	3 0
0 0 0 0	0 0 0 0	0 0 0 0	s	1 1	1 0	0 0 0 0
4	4	4	4	2	2	4

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
CALLX8 as
```

Description

CALLX8 calls subroutines using the register windows mechanism, requesting the callee rotate the window by eight registers. The CALLX8 instruction does not rotate the window itself, but instead stores the window increment for later use by the ENTRY instruction. The return address and window increment are placed in the caller's a8 (the callee's a0), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the CALLX8 instruction plus three). The window increment is also stored in the CALLINC field of the PS register, where it is accessed by the ENTRY instruction.

The target instruction address of the call is given by the contents of address register as. The target instruction must be an ENTRY instruction.

See the CALL8 instruction for calling routines where the target address is given by a PC-relative offset in the instruction.

The RETW and RETW.N (page 482) instructions return from a subroutine called by CALLX8.

The window increment stored with the return address register in a8 occupies the two most significant bits of the register, and therefore those bits must be filled in by the subroutine return. The RETW and RETW.N instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

See the CALLX0 instruction for calling routines using the non-windowed subroutine protocol.

The caller's `a8`..`a15` are the same registers as the callee's `a0`..`a7` after the callee executes the `ENTRY` instruction. You can use these registers for parameter passing. The caller's `a0`..`a7` are hidden by `CALLX8`, and therefore you may use them to keep values that are live across the call.

Operation

```
WindowCheck (00, 00, 10)
PS.CALLINC ← 10
AR[10||00] ← 10||(PC + 3)$_{29..0}$
nextPC ← AR[s]
```

Exceptions

- EveryInstR Group (see page 244)

Call Register, Rotate Window by 12 CALLX12

Instruction Word (CALLX)

23 20	19 16	15 12	11 8	7 6	5 4	3 0
0 0 0 0	0 0 0 0	0 0 0 0	s	1 1	1 1	0 0 0 0
4	4	4	4	2	2	4

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
CALLX12 as
```

Description

CALLX12 calls subroutines using the register windows mechanism, requesting the callee rotate the window by 12 registers. The CALLX12 instruction does not rotate the window itself, but instead stores the window increment for later use by the ENTRY instruction. The return address and window increment are placed in the caller’s a12 (the callee’s a0), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the CALLX12 instruction plus three). The window increment is also stored in the CALLINC field of the PS register, where it is accessed by the ENTRY instruction.

The target instruction address of the call is given by the contents of address register as. The target instruction must be an ENTRY instruction.

See the CALL12 instruction for calling routines where the target address is given by a PC-relative offset in the instruction.

The RETW and RETW.N instructions return from a subroutine called by CALLX12.

The window increment stored with the return address register in a12 occupies the two most significant bits of the register, and therefore those bits must be filled in by the subroutine return. The RETW and RETW.N instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

See the CALLX0 instruction for calling routines using the non-windowed subroutine protocol.

CALLX12 Call Register, Rotate Window by 12

The caller's a12..a15 are the same registers as the callee's a0..a3 after the callee executes the ENTRY instruction. These registers may be used for parameter passing. The caller's a0..a11 are hidden by CALLX12, and therefore may be used to keep values that are live across the call.

Operation

```
WindowCheck (00, 00, 11)
PS.CALLINC ← 11
AR[11||00] ← 11||$(PC + 3)_{29..0}$
nextPC ← AR[s]
```

Exceptions

- EveryInstR Group (see page 244)

Ceiling Single to Fixed CEIL.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 1	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
CEIL.S ar, fs, 0..15
```

Description

`CEIL.S` converts the contents of floating-point register `fs` from single-precision to signed integer format, rounding toward +∞. The single-precision value is first scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 2.0, 4.0, …, 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0` and moves to the left as `t` increases, until for `t=15` there are 15 fractional bits represented in the fixed point number. For positive overflow (value ≥ `32'h7fffffff`), positive infinity, or NaN, `32'h7fffffff` is returned; for negative overflow (value ≤ `32'h80000000`) or negative infinity, `32'h80000000` is returned. The result is written to address register `ar`.

Operation

```
AR[r] ← ceil$_s$(FR[s] ×$_s$ pow$_s$(2.0,t))
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

CLAMPS Signed Clamp

Instruction Word (RRR)

23 … 20	19 … 16	15 … 12	11 … 8	7 … 4	3 … 0
0 0 1 1	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Miscellaneous Operations Option (See Section 4.3.8 on page 62)

Assembler Syntax

```
CLAMPS ar, as, 7..22
```

Description

`CLAMPS` tests whether the contents of address register `as` fits as a signed value of `imm+1` bits (in the range 7 to 22). If so, the value is written to address register `ar`; if not, the largest value of `imm+1` bits with the same sign as `as` is written to `ar`. Thus `CLAMPS` performs the function

$$y \leftarrow \min(\max(x, -2^{imm}), 2^{imm}-1)$$

`CLAMPS` may be used in conjunction with instructions such as `ADD`, `SUB`, `MUL16S`, and so forth to implement saturating arithmetic.

Assembler Note

The immediate values accepted by the assembler are 7 to 22. The assembler encodes these in the `t` field of the instruction using 0 to 15.

Operation

```
$sign \leftarrow AR[s]_{31}$
$AR[r] \leftarrow$ if $AR[s]_{30..t+7} = sign^{24-t}$
          then $AR[s]$
          else $sign^{25-t} \| (\text{not } sign)^{t+7}$
```

Exceptions

- EveryInstR Group (see page 244)

Data Cache Hit Invalidate DHI

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 1	s	0 1 1 0	0 0 1 0
8	4	4	4	4

Required Configuration Option

Data Cache Option (See Section 4.5.5 on page 118)

Assembler Syntax

```
DHI as, 0..1020
```

Description

DHI invalidates the specified line in the level-1 data cache, if it is present. If the specified address is not in the data cache, then this instruction has no effect. If the specified address is present, it is invalidated even if it contains dirty data. If the specified line has been locked by a DPFL instruction, then no invalidation is done and no exception is raised because of the lock. The line remains in the cache and must be unlocked by a DHU or DIU instruction before it can be invalidated. This instruction is useful before a DMA write to memory that overwrites the entire line.

DHI forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises an exception (see Section 4.4.1.5 on page 89) as if it were loading from the virtual address.

Because the organization of caches is implementation-specific, the operation below specifies only a call to the implementation's dhitinval function.

DHI is a privileged instruction.

Assembler Note

To form a virtual address DHI calculates the sum of address register as and the imm8 field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     vAddr ← AR[s] + (0^22||imm8||0^2)
     (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
     if invalid(attributes) then
          EXCVADDR ← vAddr
          Exception (cause)
     else
          dhitinval(vAddr, pAddr)
     endif
endif
```

Exceptions

- EveryInstR Group (see page 244)
- Memory Group (see page 244)
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option
- GenExcep(PrivilegedCause) if Exception Option

Data Cache Hit Unlock DHU

Instruction Word (RRI4)

23 20	19 16	15 12	11 8	7 4	3 0
imm4	0 0 1 0	0 1 1 1	s	1 0 0 0	0 0 1 0
4	4	4	4	4	4

Required Configuration Option

Data Cache Index Lock Option (See Section 4.5.7 on page 122)

Assembler Syntax

```
    DHU as, 0..240
```

Description

DHU performs a data cache unlock if hit. The purpose of DHU is to remove the lock created by a DPFL instruction. Xtensa ISA implementations that do not implement cache locking must raise an illegal instruction exception when this opcode is executed.

DHU checks whether the line containing the specified address is present in the data cache, and if so, it clears the lock associated with that line. To unlock by index without knowing the address of the locked line, use the DIU instruction.

DHU forms a virtual address by adding the contents of address register as and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises an exception (see Section 4.4.1.5 on page 89) as if it were loading from the virtual address.

DHU is a privileged instruction.

Assembler Note

To form a virtual address DHU calculates the sum of address register as and the imm4 field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     vAddr ← AR[s] + ($0^{24}$||imm4||$0^4$)
     (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
     if invalid(attributes) then
          EXCVADDR ← vAddr
          Exception (cause)
     else
          dhitunlock(vAddr, pAddr)
     endif
endif
```

Exceptions

- EveryInstR Group (see page 244)
- Memory Group (see page 244)
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option
- GenExcep(PrivilegedCause) if Exception Option

Data Cache Hit Writeback DHWB

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 1	s	0 1 0 0	0 0 1 0
8	4	4	4	4

Required Configuration Option

Data Cache Option (See Section 4.5.5 on page 118)

Assembler Syntax

```
DHWB as, 0..1020
```

Description

This instruction forces dirty data in the data cache to be written back to memory. If the specified address is not in the data cache or is present but unmodified, then this instruction has no effect. If the specified address is present and modified in the data cache, the line containing it is written back, and marked unmodified. This instruction is useful before a DMA read from memory, to force writes to a frame buffer to become visible, or to force writes to memory shared by two processors.

`DHWB` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises an exception (see Section 4.4.1.5 on page 89) as if it were loading from the virtual address.

Because the organization of caches is implementation-specific, the operation below specifies only a call to the implementation's `dhitwriteback` function.

Assembler Note

To form a virtual address `DHWB` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + (0^22||imm8||0^2)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if invalid(attributes) then
     EXCVADDR ← vAddr
     Exception (cause)
else
     dhitwriteback(vAddr, pAddr)
endif
```

Exceptions

- EveryInstR Group (see page 244)
- Memory Group (see page 244)
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option

Implementation Notes

Some Xtensa ISA implementations do not support write-back caches. For these implementations, the DHWB instruction performs no operation.

Data Cache Hit Writeback Invalidate DHWBI

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 1	s	0 1 0 1	0 0 1 0
8	4	4	4	4

Required Configuration Option

Data Cache Option (See Section 4.5.5 on page 118)

Assembler Syntax

```
    DHWBI as, 0..1020
```

Description

`DHWBI` forces dirty data in the data cache to be written back to memory. If the specified address is not in the data cache, then this instruction has no effect. If the specified address is present and modified in the data cache, the line containing it is written back. After the write-back, if any, the line containing the specified address is invalidated if present. If the specified line has been locked by a `DPFL` instruction, then no invalidation is done and no exception is raised because of the lock. The line is written back but remains in the cache unmodified and must be unlocked by a `DHU` or `DIU` instruction before it can be invalidated. This instruction is useful in the same circumstances as `DHWB` and before a DMA write to memory or write from another processor to memory. If the line is certain to be completely overwritten by the write, you can use a `DHI` (as it is faster), but otherwise use a `DHWBI`.

`DHWBI` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises an exception (see Section 4.4.1.5 on page 89) as if it were loading from the virtual address.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation’s `dhitwritebackinval` function.

DHWBI Data Cache Hit Writeback Invalidate

Assembler Note

To form a virtual address, DHWBI calculates the sum of address register as and the imm8 field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^2$)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if invalid(attributes) then
     EXCVADDR ← vAddr
     Exception (cause)
else
     dhitwritebackinval(vAddr, pAddr)
endif
```

Exceptions

- EveryInstR Group (see page 244)
- Memory Group (see page 244)
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option

Implementation Notes

Some Xtensa ISA implementations do not support write-back caches. For these implementations DHWBI is identical to DHI.

Data Cache Index Invalidate DII

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 1	s	0 1 1 1	0 0 1 0
8	4	4	4	4

Required Configuration Option

Data Cache Option (See Section 4.5.5 on page 118))

Assembler Syntax

```
DII as, 0..1020
```

Description

`DII` uses the virtual address to choose a location in the data cache and invalidates the specified line. If the chosen line has been locked by a `DPFL` instruction, then no invalidation is done and no exception is raised because of the lock. The line remains in the cache and must be unlocked by a `DHU` or `DIU` instruction before it can be invalidated.The method for mapping the virtual address to a data cache location is implementation-specific. This instruction is primarily useful for data cache initialization after power-up.

`DII` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `dindexinval` function.

`DII` is a privileged instruction.

Assembler Note

To form a virtual address, `DII` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
if CRING ≠ 0 then
```

```
        Exception (PrivilegedInstructionCause)
    else
        vAddr ← AR[s] + $(0^{22}\|\mathrm{imm8}\|0^{2})$
        dindexinval(vAddr)
    endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Implementation Notes

```
    x ← ceil(log2(DataCacheBytes))
    y ← log2(DataCacheBytes ÷ DataCacheWayCount)
    z ← log2(DataCacheLineBytes)
```

The cache line specified by index Addr$_{x-1..z}$ in a direct-mapped cache or way Addr$_{x-1..y}$ and index Addr$_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache) the instruction does nothing. In some implementations all ways at index Addr$_{y-1..z}$ are invalidated regardless of the specified way, but for future compatibility this behavior should not be assumed.

The additional ways invalidated in some implementations mean that care is needed in using this instruction with write-back caches. Dirty data in any way (at the specified index) of the cache will be lost and not just dirty data in the specified way. Because the instruction is primarily used at reset, this will not usually cause any difficulty.

Data Cache Index Unlock DIU

Instruction Word (RRI4)

23 20	19 16	15 12	11 8	7 4	3 0
imm4	0 0 1 1	0 1 1 1	s	1 0 0 0	0 0 1 0
4	4	4	4	4	4

Required Configuration Option

Data Cache Index Lock Option (See Section 4.5.7 on page 122)

Assembler Syntax

```
DIU as, 0..240
```

Description

`DIU` uses the virtual address to choose a location in the data cache and unlocks the chosen line. The purpose of `DIU` is to remove the lock created by a `DPFL` instruction. The method for mapping the virtual address to a data cache location is implementation-specific. This instruction is primarily useful for unlocking the entire data cache. Xtensa ISA implementations that do not implement cache locking must raise an illegal instruction exception when this opcode is executed.

To unlock a specific cache line if it is in the cache, use the `DHU` instruction.

`DII` forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `dindexunlock` function.

`DIU` is a privileged instruction.

Assembler Note

To form a virtual address `DIU` calculates the sum of address register `as` and the `imm4` field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     vAddr ← AR[s] + $(0^{24} \| imm4 \| 0^4)$
     dindexunlock(vAddr)
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)
```

The cache line specified by index $\mathrm{Addr}_{x-1..z}$ in a direct-mapped cache or way $\mathrm{Addr}_{x-1..y}$ and index $\mathrm{Addr}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing.

Data Cache Index Write Back DIWB

Instruction Word (RRI4)

23 20	19 16	15 12	11 8	7 4	3 0
imm4	0 1 0 0	0 1 1 1	s	1 0 0 0	0 0 1 0
4	4	4	4	4	4

Required Configuration Option

Data Cache Option (See Section 4.5.5 on page 118) (added in T1050)

Assembler Syntax

```
DIWB as, 0..240
```

Description

DIWB uses the virtual address to choose a line in the data cache and writes that line back to memory if it is dirty. The method for mapping the virtual address to a data cache line is implementation-specific. This instruction is primarily useful for forcing all dirty data in the cache back to memory. If the chosen line is present but unmodified, then this instruction has no effect. If the chosen line is present and modified in the data cache, it is written back, and marked unmodified. For set-associative caches, only one line out of one way of the cache is written back. Some Xtensa ISA implementations do not support writeback caches. For these implementations DIWB does nothing.

This instruction is useful for the same purposes as DHWB, but when either the address is not known or when the range of addresses is large enough that it is faster to operate on the entire cache.

DIWB forms a virtual address by adding the contents of address register as and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's dindexwriteback function.

DIWB is a privileged instruction.

Assembler Note

To form a virtual address DIWB calculates the sum of address register as and the imm4 field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     vAddr ← AR[s] + ($0^{24}$||imm4||$0^4$)
     dindexwriteback(vAddr)
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)
```

The cache line specified by index $\text{Addr}_{x-1..z}$ in a direct-mapped cache or way $\text{Addr}_{x-1..y}$ and index $\text{Addr}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing.

Some Xtensa ISA implementations do not support write-back caches. For these implementations, the DIWB instruction has no effect.

Data Cache Index Write Back Invalidate DIWBI

Instruction Word (RRI4)

23 20	19 16	15 12	11 8	7 4	3 0
imm4	0 1 0 1	0 1 1 1	s	1 0 0 0	0 0 1 0
4	4	4	4	4	4

Required Configuration Option

Data Cache Option (See Section 4.5.5 on page 118) (added in T1050)

Assembler Syntax

```
DIWBI as, 0..240
```

Description

DIWBI uses the virtual address to choose a line in the data cache and forces that line to be written back to memory if it is dirty. After the writeback, if any, the line is invalidated. The method for mapping the virtual address to a data cache location is implementation-specific. If the chosen line is already invalid, then this instruction has no effect. If the chosen line has been locked by a DPFL instruction, then dirty data is written back but no invalidation is done and no exception is raised because of the lock. The line remains in the cache and must be unlocked by a DHU or DIU instruction before it can be invalidated. For set-associative caches, only one line out of one way of the cache is written back and invalidated. Some Xtensa ISA implementations do not support write-back caches. For these implementations DIWBI is similar to DII but invalidates only one line.

This instruction is useful for the same purposes as the DHWBI but when either the address is not known, or when the range of addresses is large enough that it is faster to operate on the entire cache.

DIWBI forms a virtual address by adding the contents of address register as and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's dindexwritebackinval function.

DIWBI is a privileged instruction.

DIWBI   Data Cache Index Write Back Invalidate

Assembler Note

To form a virtual address, DIWBI calculates the sum of address register as and the imm4 field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     vAddr ← AR[s] + ($0^{24}$||imm4||$0^{4}$)
     dindexwritebackinval(vAddr)
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)
```

The cache line specified by index $\mathrm{Addr}_{x-1..z}$ in a direct-mapped cache or way $\mathrm{Addr}_{x-1..y}$ and index $\mathrm{Addr}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing.

Data Cache Prefetch and Lock DPFL

Instruction Word (RRI4)

23 20	19 16	15 12	11 8	7 4	3 0
imm4	0 0 0 0	0 1 1 1	s	1 0 0 0	0 0 1 0
24	4	4	4	4	4

Required Configuration Option

Data Cache Index Lock Option (See Section 4.5.7 on page 122)

Assembler Syntax

```
DPFL as, 0..240
```

Description

DPFL performs a data cache prefetch and lock. The purpose of DPFL is to improve performance, and not to affect state defined by the ISA. Xtensa ISA implementations that do not implement cache locking must raise an illegal instruction exception when this opcode is executed. In general, the performance improvement from using this instruction is implementation-dependent.

DPFL checks if the line containing the specified address is present in the data cache, and if not, it begins the transfer of the line from memory to the cache. The line is placed in the data cache and the line marked as locked, that is not replaceable by ordinary data cache misses. To unlock the line, use DHU or DIU. To prefetch without locking, use the DPFR, DPFW, DPFRO, or DPFWO instructions.

DPFL forms a virtual address by adding the contents of address register as and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's dprefetch function.

DPFL is a privileged instruction.

Assembler Note

To form a virtual address, `DPFL` calculates the sum of address register `as` and the `imm4` field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     vAddr ← AR[s] + ($0^{24}$||imm4||$0^4$)
     (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
     if invalid(attributes) then
          EXCVADDR ← vAddr
          Exception (cause)
     else
          dprefetch(vAddr, pAddr, 0, 0, 1)
     endif
endif
```

Exceptions

- Memory Group (see page 244)
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option
- GenExcep(PrivilegedCause) if Exception Option

Implementation Notes

If, before the instruction executes, there are not two available DataCache ways at the required index, a Load Store Error exception is raised.

Data Cache Prefetch for Read DPFR

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 1	s	0 0 0 0	0 0 1 0
8	4	4	4	4

Required Configuration Option

Data Cache Option (See Section 4.5.5 on page 118)

Assembler Syntax

```
DPFR as, 0..1020
```

Description

DPFR performs a data cache prefetch for read. The purpose of DPFR is to improve performance, but not to affect state defined by the ISA. Therefore, some Xtensa ISA implementations may choose to implement this instruction as a simple “no-operation” instruction. In general, the performance improvement from using this instruction is implementation-dependent.

In some Xtensa ISA implementations, DPFR checks whether the line containing the specified address is present in the data cache, and if not, it begins the transfer of the line from memory. The four data prefetch instructions provide different “hints” about how the data is likely to be used in the future. DPFR indicates that the data is only likely to be read, possibly more than once, before it is replaced by another line in the cache.

DPFR forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor performs no operation. This allows the instruction to be used to speculatively fetch an address that does not exist or is protected without either causing an error or allowing inappropriate action.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation’s dprefetch function.

Assembler Note

To form a virtual address, `DPFR` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^{2}$)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if not invalid(attributes) then
     dprefetch(vAddr, pAddr, 0, 0, 0)
endif
```

Exceptions

- EveryInstR Group (see page 244)

Data Cache Prefetch for Read Once DPFRO

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 1	s	0 0 1 0	0 0 1 0
8	4	4	4	4

Required Configuration Option

Data Cache Option (See Section 4.5.5 on page 118)

Assembler Syntax

```
    DPFRO as, 0..1020
```

Description

DPFRO performs a data cache prefetch for read once. The purpose of DPFRO is to improve performance, but not to affect state defined by the ISA. Therefore, some Xtensa ISA implementations may choose to implement this instruction as a simple “no-operation” instruction. In general, the performance improvement from using this instruction is implementation-dependent.

In some Xtensa ISA implementations, DPFRO checks whether the line containing the specified address is present in the data cache, and if not, it begins the transfer of the line from memory. Four data prefetch instructions provide different “hints” about how the data is likely to be used in the future. DPFRO indicates that the data is only likely to be read once before it is replaced by another line in the cache. In some implementations, this hint might be used to select a specific cache way or to select a streaming buffer instead of the cache.

DPFRO forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor performs no operation. This allows the instruction to be used to speculatively fetch an address that does not exist or is protected without either causing an error or allowing inappropriate action.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation’s dprefetch function.

Assembler Note

To form a virtual address, DPFRO calculates the sum of address register as and the imm8 field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^{2}$)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if not invalid(attributes) then
     dprefetch(vAddr, pAddr, 0, 1, 0)
endif
```

Exceptions

- EveryInstR Group (see page 244)

Data Cache Prefetch for Write DPFW

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 1	s	0 0 0 1	0 0 1 0
8	4	4	4	4

Required Configuration Option

Data Cache Option (See Section 4.5.5 on page 118)

Assembler Syntax

```
DPFW as, 0..1020
```

Description

DPFW performs a data cache prefetch for write. The purpose of DPFW is to improve performance, but not to affect the ISA state. Therefore, some Xtensa ISA implementations may choose to implement this instruction as a simple “no-operation” instruction. In general, the performance improvement from using this instruction is implementation-dependent.

In some Xtensa ISA implementations, DPFW checks whether the line containing the specified address is present in the data cache, and if not, begins the transfer of the line from memory. Four data prefetch instructions provide different “hints” about how the data is likely to be used in the future. DPFW indicates that the data is likely to be written before it is replaced by another line in the cache. In some implementations, this fetches the data with write permission (for example, in a system with shared and exclusive states).

DPFW forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor performs no operation. This allows the instruction to be used to speculatively fetch an address that does not exist or is protected without either causing an error or allowing inappropriate action.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation’s dprefetch function.

Assembler Note

To form a virtual address `DPFW` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offsets and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^{2}$)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if not invalid(attributes) then
     dprefetch(vAddr, pAddr, 1, 0, 0)
endif
```

Exceptions

- EveryInstR Group (see page 244)

Data Cache Prefetch for Write Once DPFWO

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 1	s	0 0 1 1	0 0 1 0
8	4	4	4	4

Required Configuration Option

Data Cache Option (See Section 4.5.5 on page 118)

Assembler Syntax

```
DPFWO as, 0..1020
```

Description

`DPFWO` performs a data cache prefetch for write once. The purpose of `DPFWO` is to improve performance, but not to affect the ISA state. Therefore, some Xtensa ISA implementations may choose to implement this instruction as a simple “no-operation” instruction. In general, the performance improvement from using this instruction is implementation-dependent.

In some Xtensa ISA implementations, `DPFWO` checks whether the line containing the specified address is present in the data cache, and if not, begins the transfer of the line from memory. Four data prefetch instructions provide different “hints” about how the data is likely to be used in the future. `DPFWO` indicates that the data is likely to be read and written once before it is replaced by another line in the cache. In some implementations, this write hint fetches the data with write permission (for example, in a system with shared and exclusive states). The write-once hint might be used to select a specific cache way or to select a streaming buffer instead of the cache.

`DPFWO` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor performs no operation. This allows the instruction to be used to speculatively fetch an address that does not exist or is protected without either causing an error or allowing inappropriate action.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation’s `dprefetch` function.

Assembler Note

To form a virtual address DPFWO calculates the sum of address register as and the imm8 field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^{2}$)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if not invalid(attributes) then
    dprefetch(vAddr, pAddr, 1, 1, 0)
endif
```

Exceptions

- EveryInstR Group (see page 244)

Load/Store Synchronize DSYNC

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	0 0 1 1	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
DSYNC
```

Description

`DSYNC` waits for all previously fetched `WSR.*`, `XSR.*`, `WDTLB`, and `IDTLB` instructions to be performed before interpreting the virtual address of the next load or store instruction. This operation is also performed as part of `ISYNC`, `RSYNC`, and `ESYNC`.

This instruction is appropriate after `WSR.DBREAKC*` and `WSR.DBREAKA*` instructions. See the Special Register Tables in Section 5.3 on page 208 and Section 5.5 on page 239 for a complete description of the uses of the `DSYNC` instruction.

Because the instruction execution pipeline is implementation-specific, the operation section below specifies only a call to the implementation's `dsync` function.

Operation

```
dsync()
```

Exceptions

- EveryInst Group (see page 244)

ENTRY Subroutine Entry

Instruction Word (BRI12)

23 12	11 8	7 6	5 4	3 0
imm12	s	0 0	1 1	0 1 1 0
12	4	2	2	4

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
ENTRY as, 0..32760
```

Description

`ENTRY` is intended to be the first instruction of all subroutines called with `CALL4`, `CALL8`, `CALL12`, `CALLX4`, `CALLX8`, or `CALLX12`. This instruction is not intended to be used by a routine called by `CALL0` or `CALLX0`.

`ENTRY` serves two purposes:

1. First, it increments the register window pointer (`WindowBase`) by the amount requested by the caller (as recorded in the `PS.CALLINC` field).
2. Second, it copies the stack pointer from caller to callee and allocates the callee's stack frame. The `as` operand specifies the stack pointer register; it must specify one of `a0..a3` or the operation of `ENTRY` is undefined. It is read before the window is moved, the stack frame size is subtracted, and then the `as` register in the moved window is written.

The stack frame size is specified as the 12-bit unsigned `imm12` field in units of eight bytes. The size is zero-extended, shifted left by 3, and subtracted from the caller's stack pointer to get the callee's stack pointer. Therefore, stack frames up to 32760 bytes can be specified. The initial stack frame size must be a constant, but subsequently the `MOVSP` instruction can be used to allocate dynamically-sized objects on the stack, or to further extend a constant stack frame larger than 32760 bytes.

The windowed subroutine call protocol is described in Section 4.7.1.5 on page 187.

`ENTRY` is undefined if `PS.WOE` is `0` or if `PS.EXCM` is `1`. Some implementations raise an illegal instruction exception in these cases, as a debugging aid.

Assembler Note

In the assembler syntax, the number of bytes to be subtracted from the stack pointer is specified as the immediate. The assembler encodes this into the instruction by dividing by eight.

Operation

```
WindowCheck (00, PS.CALLINC, 00)
if as > 3 | PS.WOE = 0 | PS.EXCM = 1 then
     -- undefined operation
     -- may raise illegal instruction exception
else
     $AR[PS.CALLINC \| s_{1..0}] \leftarrow AR[s] - (0^{17} \| imm12 \| 0^{3})$
     $WindowBase \leftarrow WindowBase + (0^{2} \| PS.CALLINC)$
     $WindowStart_{WindowBase} \leftarrow 1$
endif
```

Exceptions

- EveryInstR Group (see page 244)

ESYNC Execute Synchronize

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	0 0 1 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
    ESYNC
```

Description

`ESYNC` waits for all previously fetched `WSR.*`, and `XSR.*` instructions to be performed before the next instruction uses any register values. This operation is also performed as part of `ISYNC` and `RSYNC`. `DSYNC` is performed as part of this instruction.

This instruction is appropriate after `WSR.EPC*` instructions. See the Special Register Tables in Section 5.3 on page 208 for a complete description of the uses of the `ESYNC` instruction.

Because the instruction execution pipeline is implementation-specific, the operation section below specifies only a call to the implementation's `esync` function.

Operation

```
    esync()
```

Exceptions

- EveryInst Group (see page 244)

Exception Wait EXCW

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	1 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Exception Option (See Section 4.4.1 on page 82)

Assembler Syntax

```
EXCW
```

Description

`EXCW` waits for any exceptions of previously fetched instructions to be handled. Some Xtensa ISA implementations may have imprecise exceptions; on these implementations `EXCW` waits until all previous instruction exceptions are taken or the instructions are known to be exception-free. Because the instruction execution pipeline and exception handling is implementation-specific, the operation section below specifies only a call to the implementation's `ExceptionWait` function.

Operation

```
ExceptionWait()
```

Exceptions

- EveryInst Group (see page 244)

EXTUI Extract Unsigned Immediate

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
op2	0 1 0 sa_4	r	$sae_{3..0}$	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
EXTUI ar, at, shiftimm, maskimm
```

Description

`EXTUI` performs an unsigned bit field extraction from a 32-bit register value. Specifically, it shifts the contents of address register `at` right by the shift amount `shiftimm`, which is a value `0..31` stored in bits `16` and `11..8` of the instruction word (the `sa` fields). The shift result is then ANDed with a mask of `maskimm` least-significant 1 bits and the result is written to address register `ar`. The number of mask bits, `maskimm`, may take the values `1..16`, and is stored in the `op2` field as `maskimm-1`. The bits extracted are therefore `sa+op2..sa`.

The operation of this instruction when `sa+op2` > `31` is undefined and reserved for future use.

Operation

```
$mask \leftarrow 0^{31-op2} \| 1^{op2+1}$
$AR[r] \leftarrow (0^{32} \| AR[t])_{31+sa..sa}$ and $mask$
```

Exceptions

- EveryInstR Group (see page 244)

External Wait EXTW

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	1 1 0 1	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50) (added in RA-2004.1)

Assembler Syntax

```
EXTW
```

Description

`EXTW` is a superset of `MEMW`. `EXTW` ensures that both

- all previous load, store, acquire, release, prefetch, and cache instructions; and
- any other effect of any previous instruction which is visible at the pins of the Xtensa processor

complete (or perform as described in Section 4.3.12.1 on page 74) before either

- any subsequent load, store, acquire, release, prefetch, or cache instructions; or
- external effects of the execution of any following instruction is visible at the pins of the Xtensa processor (not including instruction prefetch or TIE Queue pops)

is allowed to begin.

While `MEMW` is intended to implement the `volatile` attribute of languages such as C and C++, `EXTW` is intended to be an ordering guarantee for all external effects that the processor can have, including processor pins defined in TIE.

Because the instruction execution pipeline is implementation-specific, the operation section below specifies only a call to the implementation's `extw` function.

Operation

```
extw()
```

Exceptions

- EveryInst Group (see page 244)

FLOAT.S Convert Fixed to Single

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 0 0	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
FLOAT.S fr, as, 0..15
```

Description

`FLOAT.S` converts the contents of address register `as` from signed integer to single-precision format, rounding according to the current rounding mode. The converted integer value is then scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 0.5, 0.25, …, 1.0$\div_s$32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0` and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. The result is written to floating-point register `fr`.

Operation

```
FR[r] ← $\text{float}_s$(AR[s]) $\times_s$ $\text{pow}_s$(2.0,-t)
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Floor Single to Fixed FLOOR.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 0	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
FLOOR.S ar, fs, 0..15
```

Description

`FLOOR.S` converts the contents of floating-point register `fs` from single-precision to signed integer format, rounding toward -∞. The single-precision value is first scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 2.0, 4.0, …, 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0` and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. For positive overflow (value ≥ `32'h7fffffff`), positive infinity, or NaN, `32'h7fffffff` is returned; for negative overflow (value ≤ `32'h80000000`) or negative infinity, `32'h80000000` is returned. The result is written to address register `ar`.

Operation

```
$AR[r] \leftarrow floor_{s}(FR[s] \times_{s} pow_{s}(2.0,t))$
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

IDTLB Invalidate Data TLB Entry

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	0 0 0 0	1 1 0 0	s	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Region Protection Option (see Section 4.6.3 on page 150) or MMU Option (see Section 4.6.5 on page 158)

Assembler Syntax

```
IDTLB as
```

Description

`IDTLB` invalidates the data TLB entry specified by the contents of address register `as`. See Section 4.6 on page 138 for information on the address register formats for specific Memory Protection and Translation Options. The point at which the invalidation is effected is implementation-specific. Any translation that would be affected by this invalidation before the execution of a `DSYNC` instruction is therefore undefined.

`IDTLB` is a privileged instruction.

The representation of validity in Xtensa TLBs is implementation-specific, and thus the operation section below writes the implementation-specific value `InvalidDataTLBEntry`.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     (vpn, ei, wi) ← SplitDataTLBEntrySpec(AR[s])
     DataTLB[wi][ei] ← InvalidDataTLBEntry
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 1	s	1 1 1 0	0 0 1 0
8	4	4	4	4

Required Configuration Option

Instruction Cache Option (See Section 4.5.2 on page 115)

Assembler Syntax

```
IHI as, 0..1020
```

Description

IHI performs an instruction cache hit invalidate. It invalidates the specified line in the instruction cache, if it is present. If the specified address is not in the instruction cache, then this instruction has no effect. If the specified line is already invalid, then this instruction has no effect. If the specified line has been locked by an IPFL instruction, then no invalidation is done and no exception is raised because of the lock. The line remains in the cache and must be unlocked by an IHU or IIU instruction before it can be invalidated. Otherwise, if the specified line is present, it is invalidated.

This instruction is required before executing instructions from the instruction cache that have been written by this processor, another processor, or DMA. The writes must first be forced out of the data cache, either by using DHWB or by using stores that bypass or write through the data cache. An ISYNC instruction should then be used to guarantee that the modified instructions are visible to instruction cache misses. The instruction cache should then be invalidated for the affected addresses using a series of IHI instructions. An ISYNC instruction should then be used to guarantee that this processor's fetch pipeline does not contain instructions from the invalidated lines.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's ihitinval function.

IHI forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89). The translation is done as if the address were for an instruction fetch.

Assembler Note

To form a virtual address, `IHI` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^2$)
(pAddr, attributes, cause) ← ftranslate(vAddr, CRING)
if invalid(attributes) then
     EXCVADDR ← vAddr
     Exception (cause)
else
     ihitinval(vAddr, pAddr)
endif
```

Exceptions

- EveryInstR Group (see page 244)
- MemoryErrorException if Memory ECC/Parity Option

Instruction Cache Hit Unlock IHU

Instruction Word (RRI4)

23 20	19 16	15 12	11 8	7 4	3 0
imm4	0 0 1 0	0 1 1 1	s	1 1 0 1	0 0 1 0
4	4	4	4	4	4

Required Configuration Option

Instruction Cache Index Lock Option (See Section 4.5.4 on page 117)

Assembler Syntax

```
    IHU as, 0..240
```

Description

IHU performs an instruction cache unlock if hit. The purpose of IHU is to remove the lock created by an IPFL instruction. Xtensa ISA implementations that do not implement cache locking must raise an illegal instruction exception when this opcode is executed.

IHU checks whether the line containing the specified address is present in the instruction cache, and if so, it clears the lock associated with that line. To unlock by index without knowing the address of the locked line, use the IIU instruction.

IHU forms a virtual address by adding the contents of address register as and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example or protection violation), the processor takes one of several exceptions (see Section 4.4.1.5 on page 89). The translation is done as if the address were for an instruction fetch.

IHU is a privileged instruction.

Assembler Note

To form a virtual address, IHU calculates the sum of address register as and the imm4 field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     vAddr ← AR[s] + ($0^{24} \| imm4 \| 0^4$)
     (pAddr, attributes, cause) ← ftranslate(vAddr, CRING)
     if invalid(attributes) then
          EXCVADDR ← vAddr
          Exception (cause)
     else
          ihitunlock(vAddr, pAddr)
     endif
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option
- MemoryErrorException if Memory ECC/Parity Option

Instruction Cache Index Invalidate III

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 1	s	1 1 1 1	0 0 1 0
8	4	4	4	4

Required Configuration Option

Instruction Cache Option (See Section 4.5.2 on page 115)

Assembler Syntax

```
III as, 0..1020
```

Description

`III` performs an instruction cache index invalidate. This instruction uses the virtual address to choose a location in the instruction cache and invalidates the specified line. The method for mapping the virtual address to an instruction cache location is implementation-specific. If the chosen line is already invalid, then this instruction has no effect. If the chosen line has been locked by an `IPFL` instruction, then no invalidation is done and no exception is raised because of the lock. The line remains in the cache and must be unlocked by an `IHU` or `IIU` instruction before it can be invalidated. This instruction is useful for instruction cache initialization after power-up or for invalidating the entire instruction cache.

`III` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `iindexinval` function.

`III` is a privileged instruction.

Assembler Note

To form a virtual address, `III` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     $vAddr$ ← AR[s] + ($0^{22}$‖imm8‖$0^2$)
     iindexinval($vAddr$, $pAddr$)
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Implementation Notes

```
x ← ceil(log2(InstCacheBytes))
y ← log2(InstCacheBytes ÷ InstCacheWayCount)
z ← log2(InstCacheLineBytes)
```

The cache line specified by index $\mathrm{Addr}_{x-1..z}$ in a direct-mapped cache or way $\mathrm{Addr}_{x-1..y}$ and index $\mathrm{Addr}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing. In some implementations all ways at index $\mathrm{Addr}_{y-1..z}$ are invalidated regardless of the specified way, but for future compatibility this behavior should not be assumed.

Invalidate Instruction TLB Entry IITLB

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	0 0 0 0	0 1 0 0	s	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Region Protection Option (see Section 4.6.3 on page 150) or MMU Option (see Section 4.6.5 on page 158)

Assembler Syntax

```
IITLB as
```

Description

`IITLB` invalidates the instruction TLB entry specified by the contents of address register `as`. See Section 4.6 on page 138 for information on the address register formats for specific Memory Protection and Translation options. The point at which the invalidation is effected is implementation-specific. Any translation that would be affected by this invalidation before the execution of an `ISYNC` instruction is therefore undefined.

`IITLB` is a privileged instruction.

The representation of validity in Xtensa TLBs is implementation-specific, and thus the operation section below writes the implementation-specific value `InvalidInstTLBEntry`.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     (vpn, ei, wi) ← SplitInstTLBEntrySpec(AR[s])
     InstTLB[wi][ei] ← InvalidInstTLBEntry
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

IIU Instruction Cache Index Unlock

Instruction Word (RRI4)

23 20	19 16	15 12	11 8	7 4	3 0
imm4	0 0 1 1	0 1 1 1	s	1 1 0 1	0 0 1 0
4	4	4	4	4	4

Required Configuration Option

Instruction Cache Index Lock Option (See Section 4.5.4 on page 117)

Assembler Syntax

```
IIU as, 0..240
```

Description

`IIU` uses the virtual address to choose a location in the instruction cache and unlocks the chosen line. The purpose of `IIU` is to remove the lock created by an `IPFL` instruction. The method for mapping the virtual address to an instruction cache location is implementation-specific. This instruction is primarily useful for unlocking the entire instruction cache. Xtensa ISA implementations that do not implement cache locking must raise an illegal instruction exception when this opcode is executed.

To unlock a specific cache line if it is in the cache, use the `IHU` instruction.

`IIU` forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `iindexunlock` function.

`IIU` is a privileged instruction.

Assembler Note

To form a virtual address `IIU` calculates the sum of address register `as` and the `imm4` field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     vAddr ← AR[s] + ($0^{24}$||imm4||$0^4$)
     iindexunlock(vAddr)
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(InstCacheBytes))
y ← log2(InstCacheBytes ÷ InstCacheWayCount)
z ← log2(InstCacheLineBytes)
```

The cache line specified by index $\text{Addr}_{x-1..z}$ in a direct-mapped cache or way $\text{Addr}_{x-1..y}$ and index $\text{Addr}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing.

ILL Illegal Instruction

Instruction Word (CALLX)

23 20	19 16	15 12	11 8	7 6	5 4	3 0
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0	0 0	0 0 0 0
4	4	4	4	2	2	4

Required Configuration Option

Exception Option (See Section 4.4.1 on page 82)

Assembler Syntax

```
ILL
```

Description

`ILL` is an opcode that is guaranteed to raise an illegal instruction exception in all implementations.

Operation

```
Exception(IllegalInstructionCause)
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(IllegalInstructionCause) if Exception Option

Narrow Illegal Instruction ILL.N

Instruction Word (RRRN)

15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 0	0 1 1 0	1 1 0 1
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53) and Exception Option (See Section 4.4.1 on page 82)

Assembler Syntax

```
ILL.N
```

Description

ILL.N is a 16-bit opcode that is guaranteed to raise an illegal instruction exception.

Operation

```
Exception(IllegalInstructionCause)
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(IllegalInstructionCause) if Exception Option

IPF Instruction Cache Prefetch

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 1	s	1 1 0 0	0 0 1 0
8	4	4	4	4

Required Configuration Option

Instruction Cache Option (See Section 4.5.2 on page 115)

Assembler Syntax

```
IPF as, 0..1020
```

Description

`IPF` performs an instruction cache prefetch. The purpose of `IPF` is to improve performance, but not to affect state defined by the ISA. Therefore, some Xtensa ISA implementations may choose to implement this instruction as a simple “no-operation” instruction. In general, the performance improvement from using this instruction is implementation-dependent. In some implementations, `IPF` checks whether the line containing the specified address is present in the instruction cache, and if not, it begins the transfer of the line from memory to the instruction cache. Prefetching an instruction line may prevent the processor from taking an instruction cache miss later.

`IPF` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation, or non-existent memory), the processor performs no operation. This allows the instruction to be used to speculatively fetch an address that does not exist or is protected without either causing an error or allowing inappropriate action. The translation is done as if the address were for an instruction fetch.

Assembler Note

To form a virtual address, `IPF` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^{2}$)
(pAddr, attributes, cause) ← ftranslate(vAddr, CRING)
if not invalid(attributes) then
     iprefetch(vAddr, pAddr, 0)
endif
```

Exceptions

- EveryInstR Group (see page 244)

IPFL Instruction Cache Prefetch and Lock

Instruction Word (RRI4)

23 20	19 16	15 12	11 8	7 4	3 0
imm4	0 0 0 0	0 1 1 1	s	1 1 0 1	0 0 1 0
4	4	4	4	4	4

Required Configuration Option

Instruction Cache Index Lock Option (See Section 4.5.4 on page 117)

Assembler Syntax

```
IPFL as, 0..240
```

Description

IPFL performs an instruction cache prefetch and lock. The purpose of IPFL is to improve performance, but not to affect state defined by the ISA. Xtensa ISA implementations that do not implement cache locking must raise an illegal instruction exception when this opcode is executed. In general, the performance improvement from using this instruction is implementation-dependent as implementations may not overlap the cache fill with the execution of other instructions.

In some implementations, IPFL checks whether the line containing the specified address is present in the instruction cache, and if not, begins the transfer of the line from memory to the instruction cache. The line is placed in the instruction cache and marked as locked, so it is not replaceable by ordinary instruction cache misses. To unlock the line, use IHU or IIU. To prefetch without locking, use the IPF instruction.

IPFL forms a virtual address by adding the contents of address register as and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89). The translation is done as if the address were for an instruction fetch. If the line cannot be cached, an exception is raised with cause InstructionFetchErrorCause.

IPFL is a privileged instruction.

Assembler Note

To form a virtual address, `IPFL` calculates the sum of address register `as` and the `imm4` field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     vAddr ← AR[s] + (0^24||imm4||0^4)
     (pAddr, attributes, cause) ← ftranslate(vAddr, CRING)
     if invalid(attributes) then
          EXCVADDR ← vAddr
          Exception (cause)
     else
          iprefetch(vAddr, pAddr, 1)
     endif
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Implementation Notes

If there are not two available InstCache ways at the required index before the instruction executes, an exception is raised.

ISYNC Instruction Fetch Synchronize

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
ISYNC
```

Description

`ISYNC` waits for all previously fetched load, store, cache, TLB, `WSR.*`, and `XSR.*` instructions that affect instruction fetch to be performed before fetching the next instruction. `RSYNC`, `ESYNC`, and `DSYNC` are performed as part of this instruction.

The proper sequence for writing instructions and then executing them is:

- write instructions
- use `DHWB` to force the data out of the data cache (this step may be skipped if write-through, bypass, or no allocate stores were used)
- use `ISYNC` to wait for the writes to be visible to instruction cache misses
- use multiple `IHI` instructions to invalidate the instruction cache for any lines that were modified (this step is not appropriate if the affected instructions are in InstRAM or cannot be cached)
- use `ISYNC` to ensure that fetch pipeline will see the new instructions

This instruction also waits for all previously executed `WSR.*` and `XSR.*` instructions that affect instruction fetch or register access processor state, including:

- `WSR.LCOUNT`, `WSR.LBEG`, `WSR.LEND`
- `WSR.IBREAKENABLE`, `WSR.IBREAKA[i]`
- `WSR.CCOMPAREn`

See the Special Register Tables in Section 5.3 on page 208 and Section 5.7 on page 240, for a complete description of the `ISYNC` instruction's uses.

Operation

```
isync()
```

Exceptions

- EveryInst Group (see page 244)

Implementation Notes

In many implementations, `ISYNC` consumes considerably more cycles than `RSYNC`, `ESYNC`, or `DSYNC`.

J Unconditional Jump

Instruction Word (CALL)

23 6	5	4	3			0
offset	0	0	0	1	1	0
18	2		4			

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
J label
```

Description

`J` performs an unconditional branch to the target address. It uses a signed, 18-bit PC-relative offset to specify the target address. The target address is given by the address of the `J` instruction plus the sign-extended 18-bit `offset` field of the instruction plus four, giving a range of −`131068` to `+131075` bytes.

Operation

$$\text{nextPC} \leftarrow \text{PC} + (\text{offset}_{17}{}^{14} \| \text{offset}) + 4$$

Exceptions

- EveryInst Group (see page 244)

Unconditional Jump Long J.L

Instruction Word (CALL)

23	6	5	4	3			0
offset		0	0	0	1	1	0
18		2		4			

Required Configuration Option

Assembler Macro

Assembler Syntax

```
J.L label, an
```

Description

`J.L` is an assembler macro which generates exactly a `J` instruction as long as the offset will reach the label. If the offset is not long enough, the assembler relaxes the instruction to a literal load into `an` followed by a `JX an`.. The AR register `an` may or may not be modified.

Exceptions

- EveryInstR Group (see page 244)

JX Unconditional Jump Register

Instruction Word (CALLX)

23 20	19 16	15 12	11 8	7 6	5 4	3 0
0 0 0 0	0 0 0 0	0 0 0 0	s	1 0	1 0	0 0 0 0
4	4	4	4	2	2	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
JX as
```

Description

`JX` performs an unconditional jump to the address in register `as`.

Operation

```
nextPC ← AR[s]
```

Exceptions

- EveryInstR Group (see page 244)

Load 8-bit Unsigned L8UI

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 0 0 0	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
L8UI at, as, 0..255
```

Description

`L8UI` is an 8-bit unsigned load from memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word. Therefore, the offset ranges from 0 to 255. Eight bits (one byte) are read from the physical address. This data is then zero-extended and written to address register `at`.

If the Region Translation Option (page 156) or the MMU Option (page 158)is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Operation

```
vAddr ← AR[s] + ($0^{24}$||imm8)
(mem8, error) ← Load8(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     AR[t] ← $0^{24}$||mem8
endif
```

Exceptions

- Memory Group (see page 244)
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option
- DebugExcep(DBREAK) if Debug Option

L16SI Load 16-bit Signed

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 0 0 1	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
L16SI at, as, 0..510
```

Description

`L16SI` is a 16-bit signed load from memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by 1. Therefore, the offset can specify multiples of two from zero to 510. Sixteen bits (two bytes) are read from the physical address. This data is then sign-extended and written to address register `at`.

If the Region Translation Option (page 156) or the MMU Option (page 158)is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation, non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the least significant address bit is ignored; a reference to an odd address produces the same result as a reference to the address minus one. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, `L16SI` calculates the sum of address register `as` and the `imm8` field of the instruction word times two. Therefore, the machine-code offset is in terms of 16-bit (2 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by two.

Operation

```
vAddr ← AR[s] + ($0^{23}$||imm8||0)
(mem16, error) ← Load16(vAddr)
```

```
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     AR[t] ← $mem16_{15}^{16} \| mem16$
endif
```

Exceptions

- Memory Load Group (see page 244)

L16UI Load 16-bit Unsigned

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 0 0 1	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
L16UI at, as, 0..510
```

Description

`L16UI` is a 16-bit unsigned load from memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by 1. Therefore, the offset can specify multiples of two from zero to 510. Sixteen bits (two bytes) are read from the physical address. This data is then zero-extended and written to address register `at`.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the least significant address bit is ignored; a reference to an odd address produces the same result as a reference to the address minus one. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, `L16UI` calculates the sum of address register `as` and the `imm8` field of the instruction word times two. Therefore, the machine-code offset is in terms of 16-bit (2 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by two.

Operation

```
vAddr ← AR[s] + ($0^{23}$||imm8||0)
(mem16, error) ← Load16(vAddr)
```

```
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     AR[t] ← $0^{16}$||mem16
endif
```

Exceptions

- Memory Load Group (see page 244)

L32AI — Load 32-bit Acquire

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 0 1 1	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Multiprocessor Synchronization Option (See Section 4.3.12 on page 74)

Assembler Syntax

```
L32AI at, as, 0..1020
```

Description

L32AI is a 32-bit load from memory with “acquire” semantics. This load performs before any subsequent loads, stores, acquires, or releases are performed. It is typically used to test a synchronization variable protecting a critical region (for example, to acquire a lock).

L32AI forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. 32 bits (four bytes) are read from the physical address. This data is then written to address register at. L32AI causes the processor to delay processing of subsequent loads, stores, acquires, and releases until the L32AI is performed. In some Xtensa ISA implementations, this occurs automatically and L32AI is identical to L32I. Other implementations (for example, those with multiple outstanding loads and stores) delay processing as described above. Because the method of delay is implementation-dependent, this is indicated in the operation section below by the implementation function acquire.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, `L32AI` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^{2}$)
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     AR[t] ← mem32
     acquire()
endif
```

Exceptions

- Memory Load Group (see page 244)

L32E Load 32-bit for Window Exceptions

Instruction Word (RRI4)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	1 0 0 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
    L32E at, as, -64..-4
```

Description

L32E is a 32-bit load instruction similar to L32I but with semantics required by window overflow and window underflow exception handlers. In particular, memory access checking is done with PS.RING instead of CRING, and the offset used to form the virtual address is a 4-bit one-extended immediate. Therefore, the offset can specify multiples of four from -64 to -4. In configurations without the MMU Option, there is no PS.RING, and L32E is similar to L32I with a negative offset.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

L32E is a privileged instruction.

Assembler Note

To form a virtual address, L32E calculates the sum of address register as and the r field of the instruction word times four (and one extended). Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Load 32-bit for Window Exceptions L32E

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     vAddr ← AR[s] + ($1^{26} \| r \| 0^{2}$)
     ring ← if MMU Option then PS.RING else 0
     (mem32, error) ← Load32Ring(vAddr, ring)
     if error then
          EXCVADDR ← vAddr
          Exception (LoadStoreErrorCause)
     else
          AR[t] ← mem32
     endif
endif
```

Exceptions

- Memory Load Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

L32I Load 32-bit

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 0 1 0	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
L32I at, as, 0..1020
```

Description

L32I is a 32-bit load from memory. It forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. Thirty-two bits (four bytes) are read from the physical address. This data is then written to address register at.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation, non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

L32I is one of only a few memory reference instructions that can access instruction RAM/ROM.

Assembler Note

The assembler may convert L32I instructions to L32I.N when the Code Density Option is enabled and the immediate operand falls within the available range. Prefixing the L32I instruction with an underscore (_L32I) disables this optimization and forces the assembler to generate the wide form of the instruction.

Load 32-bit

To form a virtual address, `L32I` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + (0^22||imm8||0^2)
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     AR[t] ← mem32
endif
```

Exceptions

- Memory Load Group (see page 244)

L32I.N Narrow Load 32-bit

Instruction Word (RRRN)

15 12	11 8	7 4	3 0
imm4	s	t	1 0 0 0
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53)

Assembler Syntax

```
L32I.N at, as, 0..60
```

Description

`L32I.N` is similar to `L32I`, but has a 16-bit encoding and supports a smaller range of offset values encoded in the instruction word.

`L32I.N` is a 32-bit load from memory. It forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 60. Thirty-two bits (four bytes) are read from the physical address. This data is then written to address register `at`.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

`L32I.N` is one of only a few memory reference instructions that can access instruction RAM/ROM.

Narrow Load 32-bit L32I.N

Assembler Note

The assembler may convert L32I.N instructions to L32I. Prefixing the L32I.N instruction with an underscore (_L32I.N) disables this optimization and forces the assembler to generate the narrow form of the instruction.

To form a virtual address, L32I.N calculates the sum of address register as and the imm4 field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{26}$||imm4||$0^{2}$)
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    AR[t] ← mem32
endif
```

Exceptions

- Memory Load Group (see page 244)

L32R Load 32-bit PC-Relative

Instruction Word (RI6)

23 8	7 4	3 0
imm16	t	0 0 0 1
16	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
L32R at, label
```

Description

`L32R` is a PC-relative 32-bit load from memory. It is typically used to load constant values into a register when the constant cannot be encoded in a `MOVI` instruction.

`L32R` forms a virtual address by adding the 16-bit one-extended constant value encoded in the instruction word shifted left by two to the address of the `L32R` plus three with the two least significant bits cleared. Therefore, the offset can always specify 32-bit aligned addresses from -262141 to -4 bytes from the address of the `L32R` instruction. 32 bits (four bytes) are read from the physical address. This data is then written to address register `at`.

In the presence of the Extended L32R Option (Section 4.3.3 on page 56) when `LITBASE[0]` is clear, the instruction has the identical operation. When `LITBASE[0]` is set, `L32R` forms a virtual address by adding the 16-bit one extended constant value encoded in the instruction word shifted left by two to the literal base address indicated by the upper 20 bits of `LITBASE`. The offset can specify 32-bit aligned addresses from -262144 to -4 bytes from the literal base address.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

It is not possible to specify an unaligned address.

`L32R` is one of only a few memory reference instructions that can access instruction RAM/ROM.

Assembler Note

In the assembler syntax, the immediate operand is specified as the address of the location to load from, rather than the offset from the current instruction address. The linker and the assembler both assume that the location loaded by the `L32R` instruction has not been and will not be accessed by any other type of load or store instruction and optimizes according to that assumption.

Operation

```
if Extended L32R Option and $\text{LITBASE}_0$ then
     vAddr ← $(\text{LITBASE}_{31..12} \| 0^{12}) + (1^{14} \| \text{imm16} \| 0^2)$
else
     vAddr ← $((\text{PC} + 3)_{31..2} \| 0^2) + (1^{14} \| \text{imm16} \| 0^2)$
endif
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     AR[t] ← mem32
endif
```

Exceptions

- Memory Group (see page 244)
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option
- DebugExcep(DBREAK) if Debug Option

LDCT

Load Data Cache Tag

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 1	1 0 0 0	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Data Cache Test Option (See Section 4.5.6 on page 121)

Assembler Syntax

```
LDCT at, as
```

Description

LDCT is not part of the Xtensa Instruction Set Architecture, but is instead specific to an implementation. That is, it may not exist in all implementations of the Xtensa ISA.

LDCT is intended for reading the RAM array that implements the data cache tags as part of manufacturing test.

LDCT uses the contents of address register as to select a line in the data cache, reads the tag associated with this line, and writes the result to address register at. The value written to at is described under Cache Tag Format in Section 4.5.1.2 on page 112.

LDCT is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     index ← AR[s]$_{dih..dil}$
     AR[t] ← DataCacheTag[index]
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)
```

The cache line specified by index AR[s]$_{x-1..z}$ in a direct-mapped cache or way AR[s]$_{x-1..y}$ and index AR[s]$_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction loads an undefined value.

LDDEC Load with Autodecrement

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 1	0 0 0 0	0 0 w	s	0 0 0 0	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
LDDEC mw, as
```

Description

`LDDEC` loads MAC16 register `mw` from memory using auto-decrement addressing. It forms a virtual address by subtracting 4 from the contents of address register `as`. 32 bits (four bytes) are read from the physical address. This data is then written to MAC16 register `mw`, and the virtual address is written back to address register `as`.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

```
vAddr ← AR[s] − 4
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     MR[w] ← mem32
     AR[s] ← vAddr
endif
```

Exceptions

- Memory Load Group (see page 244)

Load with Autoincrement LDINC

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 0	0 0 0 0	0 0 w	s	0 0 0 0	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
LDINC mw, as
```

Description

`LDINC` loads MAC16 register `mw` from memory using auto-increment addressing. It forms a virtual address by adding 4 to the contents of address register `as`. 32 bits (four bytes) are read from the physical address. This data is then written to MAC16 register `mw`, and the virtual address is written back to address register `as`.

If the Region Translation Option (page 156) or the MMU Option (page 158)is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

```
vAddr ← AR[s] + 4
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     MR[w] ← mem32
     AR[s] ← vAddr
endif
```

Exceptions

- Memory Load Group (see page 244)

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 1	0 0 0 0	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Instruction Cache Test Option (See Section 4.5.3 on page 116)

Assembler Syntax

```
LICT at, as
```

Description

LICT is not part of the Xtensa Instruction Set Architecture, but is instead specific to an implementation. That is, it may not exist in all implementations of the Xtensa ISA.

LICT is intended for reading the RAM array that implements the instruction cache tags as part of manufacturing test.

LICT uses the contents of address register as to select a line in the instruction cache, reads the tag associated with this line, and writes the result to address register at. The value written to at is described under Cache Tag Format in Section 4.5.1.2 on page 112.

LICT is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     index ← AR[s]$_{iih..iil}$
     AR[t] ← InstCacheTag[index]
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Implementation Notes

```
x ← ceil(log2(InstCacheBytes))
y ← log2(InstCacheBytes ÷ InstCacheWayCount)
z ← log2(InstCacheLineBytes)
```

The cache line specified by index $\texttt{AR[s]}_{\texttt{x-1..z}}$ in a direct-mapped cache or way $\texttt{AR[s]}_{\texttt{x-1..y}}$ and index $\texttt{AR[s]}_{\texttt{y-1..z}}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction loads an undefined value.

LICW Load Instruction Cache Word

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 1	0 0 1 0	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Instruction Cache Test Option (See Section 4.5.3 on page 116)

Assembler Syntax

```
LICW at, as
```

Description

LICW is not part of the Xtensa Instruction Set Architecture, but is instead specific to an implementation. That is, it may not exist in all implementations of the Xtensa ISA.

LICW is intended for reading the RAM array that implements the instruction cache as part of manufacturing test.

LICW uses the contents of address register as to select a line in the instruction cache and one 32-bit quantity within that line, reads that data, and writes the result to address register at.

LICW is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     index ← $AR[s]_{iih..2}$
     AR[t] ← InstCacheData [index]
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(InstCacheBytes))
y ← log2(InstCacheBytes ÷ InstCacheWayCount)
z ← log2(InstCacheLineBytes)
```

The cache line specified by index $\texttt{AR[s]}_{x-1..z}$ in a direct-mapped cache or way $\texttt{AR[s]}_{x-1..y}$ and index $\texttt{AR[s]}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction loads an undefined value. Within the cache line, $\texttt{AR[s]}_{z-1..2}$ is used to determine which 32-bit quantity within the line is loaded.

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 0 0 0	s	0 1 1 1	0 1 1 0
8	4	4	4	4

Required Configuration Option

Loop Option (See Section 4.3.2 on page 54)

Assembler Syntax

```
LOOP as, label
```

Description

LOOP sets up a zero-overhead loop by setting the LCOUNT, LBEG, and LEND special registers, which control instruction fetch. The loop will iterate the number of times specified by address register as, with 0 causing the loop to iterate 2^{32} times. LCOUNT, the current loop iteration counter, is loaded from the contents of address register as minus one. LEND is the loop end address and is loaded with the address of the LOOP instruction plus four, plus the zero-extended 8-bit offset encoded in the instruction (therefore, the loop code may be up to 256 bytes in length). LBEG, the loop begin address, is loaded with the address of the following instruction (the address of the LOOP instruction plus three).

After the processor fetches an instruction that increments the PC to the value contained in LEND, and LCOUNT is not zero, it loads the PC with the contents of LBEG and decrements LCOUNT. LOOP is intended to be implemented with help from the instruction fetch engine of the processor, and therefore should not incur a mispredict or taken branch penalty. Branches and jumps to the address contained in LEND do not cause a loop back, and therefore may be used to exit the loop prematurely. Likewise, a return from a call instruction as the last instruction of the loop would not trigger loop back; this case should be avoided.

There is no mechanism to proceed to the next iteration of the loop from the middle of the loop. The compiler may insert a branch to a NOP placed as the last instruction of the loop to implement this function if required.

Because LCOUNT, LBEG, and LEND are single registers, zero-overhead loops may not be nested. Using conditional branch instructions to implement outer level loops is typically not a performance issue. Because loops cannot be nested, it is usually inappropriate to include a procedure call inside a loop (the callee might itself use a zero-overhead loop).

To simplify the implementation of zero-overhead loops, the LBEG address, which is the LOOP instruction address plus three, must be such that the first instruction must entirely fit within a naturally aligned four byte region or, if the instruction is larger than four bytes, a naturally aligned region which is the next power of two equal to or larger than the instruction. When the LOOP instruction would not naturally be placed at such an address, the insertion of NOP instructions or adjustment of which instructions are 16-bit density instructions is sufficient to give it the required alignment.

The automatic loop-back when the PC increments to match LEND is disabled when PS.EXCM is set. This prevents non-privileged code from affecting the operation of the privileged exception vector code.

Assembler Note

The assembler automatically aligns the LOOP instruction as required.

When the label is out of range, the assembler may insert a number of instructions to extend the size of the loop. Prefixing the instruction mnemonic with an underscore (_LOOP) disables this feature and forces the assembler to generate an error in this case.

Operation

```
LCOUNT ← AR[s] − 1
LBEG ← PC + 3
LEND ← PC + ($0^{24}$||imm8) + 4
```

Exceptions

- EveryInstR Group (see page 244)

Implementation Notes

In some implementations, LOOP takes an extra clock for the first loop back of certain loops. In addition, certain instructions (such as ISYNC or a write to LEND) may cause an additional cycle on the following loop back.

LOOPGTZ Loop if Greater Than Zero

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 0 1 0	s	0 1 1 1	0 1 1 0
8	4	4	4	4

Required Configuration Option

Loop Option (See Section 4.3.2 on page 54)

Assembler Syntax

```
LOOPGTZ as, label
```

Description

LOOPGTZ sets up a zero-overhead loop by setting the LCOUNT, LBEG, and LEND special registers, which control instruction fetch. The loop will iterate the number of times specified by address register as with values ≤ 0 causing the loop to be skipped altogether by branching directly to the loop end address. LCOUNT, the current loop iteration counter, is loaded from the contents of address register as minus one. LEND is the loop end address and is loaded with the address of the LOOPGTZ instruction plus four, plus the zero-extended 8-bit offset encoded in the instruction (therefore, the loop code may be up to 256 bytes in length). LBEG, the loop begin address, is loaded with the address of the following instruction (the address of the LOOPGTZ instruction plus three). LCOUNT, LEND, and LBEG are still loaded even when the loop is skipped.

After the processor fetches an instruction that increments the PC to the value contained in LEND, and LCOUNT is not zero, it loads the PC with the contents of LBEG and decrements LCOUNT. LOOPGTZ is intended to be implemented with help from the instruction fetch engine of the processor, and therefore should not incur a mispredict or taken branch penalty. Branches and jumps to the address contained in LEND do not cause a loop back, and therefore may be used to exit the loop prematurely. Similarly, a return from a call instruction as the last instruction of the loop would not trigger loop back; this case should be avoided.

There is no mechanism to proceed to the next iteration of the loop from the middle of the loop. The compiler may insert a branch to a NOP placed as the last instruction of the loop to implement this function if required.

Because LCOUNT, LBEG, and LEND are single registers, zero-overhead loops may not be nested. Using conditional branch instructions to implement outer level loops is typically not a performance issue. Because loops cannot be nested, it is usually inappropriate to include a procedure call inside a loop (the callee might itself use a zero-overhead loop).

To simplify the implementation of zero-overhead loops, the LBEG address, which is the LOOP instruction address plus three, must be such that the first instruction must entirely fit within a naturally aligned four byte region or, if the instruction is larger than four bytes, a naturally aligned region which is the next power of two equal to or larger than the instruction. When the LOOP instruction would not naturally be placed at such an address, the insertion of NOP instructions or adjustment of which instructions are 16-bit density instructions is sufficient to give it the required alignment.

The automatic loop-back when the PC increments to match LEND is disabled when PS.EXCM is set. This prevents non-privileged code from affecting the operation of the privileged exception vector code.

Assembler Note

The assembler automatically aligns the LOOPGTZ instruction as required.

When the label is out of range, the assembler may insert a number of instructions to extend the size of the loop. Prefixing the instruction mnemonic with an underscore (_LOOPGTZ) disables this feature and forces the assembler to generate an error in this case.

Operation

```
LCOUNT ← AR[s] − 1
LBEG ← PC + 3
LEND ← PC + ($0^{24}$||imm8) + 4
if AR[s] ≤ $0^{32}$ then
     nextPC ← PC + ($0^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Implementation Notes

In some implementations, LOOPGTZ takes an extra clock for the first loop back of certain loops. In addition, certain instructions (such as ISYNC or a write to LEND) may cause an additional cycle on the following loop back.

LOOPNEZ Loop if Not-Equal Zero

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 0 0 1	s	0 1 1 1	0 1 1 0
8	4	4	4	4

Required Configuration Option

Loop Option (See Section 4.3.2 on page 54)

Assembler Syntax

```
LOOPNEZ as, label
```

Description

LOOPNEZ sets up a zero-overhead loop by setting the LCOUNT, LBEG, and LEND special registers, which control instruction fetch. The loop will iterate the number of times specified by address register as with the zero value causing the loop to be skipped altogether by branching directly to the loop end address. LCOUNT, the current loop iteration counter, is loaded from the contents of address register as minus 1. LEND is the loop end address and is loaded with the address of the LOOPNEZ instruction plus four plus the zero-extended 8-bit offset encoded in the instruction (therefore, the loop code may be up to 256 bytes in length). LBEG is loaded with the address of the following instruction (the address of the LOOPNEZ instruction plus three). LCOUNT, LEND, and LBEG are still loaded even when the loop is skipped.

After the processor fetches an instruction that increments the PC to the value contained in LEND, and LCOUNT is not zero, it loads the PC with the contents of LBEG and decrements LCOUNT. LOOPNEZ is intended to be implemented with help from the instruction fetch engine of the processor, and therefore should not incur a mispredict or taken branch penalty. Branches and jumps to the address contained in LEND do not cause a loop back, and therefore may be used to exit the loop prematurely. Similarly a return from a call instruction as the last instruction of the loop would not trigger loop back; this case should be avoided.

There is no mechanism to proceed to the next iteration of the loop from the middle of the loop. The compiler may insert a branch to a NOP placed as the last instruction of the loop to implement this function if required.

Because LCOUNT, LBEG, and LEND are single registers, zero-overhead loops may not be nested. Using conditional branch instructions to implement outer level loops is typically not a performance issue. Because loops cannot be nested, it is usually inappropriate to include a procedure call inside a loop (the callee might itself use a zero-overhead loop).

To simplify the implementation of zero-overhead loops, the LBEG address, which is the LOOP instruction address plus three, must be such that the first instruction must entirely fit within a naturally aligned four byte region or, if the instruction is larger than four bytes, a naturally aligned region which is the next power of two equal to or larger than the instruction. When the LOOP instruction would not naturally be placed at such an address, the insertion of NOP instructions or adjustment of which instructions are 16-bit density instructions is sufficient to give it the required alignment.

The automatic loop-back when the PC increments to match LEND is disabled when PS.EXCM is set. This prevents non-privileged code from affecting the operation of the privileged exception vector code.

Assembler Note

The assembler automatically aligns the LOOPNEZ instruction as required.

When the label is out of range, the assembler may insert a number of instructions to extend the size of the loop. Prefixing the instruction mnemonic with an underscore (_LOOPNEZ) disables this feature and forces the assembler to generate an error in this case.

Operation

```
LCOUNT ← AR[s] − 1
LBEG ← PC + 3
LEND ← PC + ($0^{24}$||imm8) + 4)
if AR[s] = $0^{32}$ then
     nextPC ← PC + ($0^{24}$||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see page 244)

Implementation Notes

In some implementations, LOOPNEZ takes an extra clock for the first loop back of certain loops. In addition, certain instructions (such as ISYNC or a write to LEND) may cause an additional cycle on the following loop back.

LSI Load Single Immediate

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 0 0 0	s	t	0 0 1 1
8	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
LSI ft, as, 0..1020
```

Description

`LSI` is a 32-bit load from memory to the floating-point register file. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. Thirty-two bits (four bytes) are read from the physical address. This data is then written to floating-point register `ft`.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, `LSI` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^{2}$)
(mem32, error) ← Load32(vAddr)
```

```
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     FR[t] ← mem32
endif
```

Exceptions

- Memory Load Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

LSIU Load Single Immediate with Update

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 0 0 0	s	t	0 0 1 1
8	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
LSIU ft, as, 0..1020
```

Description

LSIU is a 32-bit load from memory to the floating-point register file with base address register update. It forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. Thirty-two bits (four bytes) are read from the physical address. This data is then written to floating-point register ft and the virtual address is written back to address register as.

If the Region Translation Option (page 156) or the MMU Option (page 158)is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, LSIU calculates the sum of address register as and the imm8 field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
$vAddr \leftarrow \mathtt{AR[s]} + (0^{22}\|\mathtt{imm8}\|0^{2})$
```

```
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     FR[t] ← mem32
     AS[s] ← vAddr
endif
```

Exceptions

- Memory Load Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

LSX Load Single Indexed

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	1 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
LSX fr, as, at
```

Description

`LSX` is a 32-bit load from memory to the floating-point register file. It forms a virtual address by adding the contents of address register `as` and the contents of address register `at`. 32 bits (four bytes) are read from the physical address. This data is then written to floating-point register `fr`.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

```
vAddr ← AR[s] + (AR[t])
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     FR[r] ← mem32
endif
```

Exceptions

- Memory Load Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

LSXU Load Single Indexed with Update

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 1	1 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
LSXU fr, as, at
```

Description

`LSXU` is a 32-bit load from memory to the floating-point register file with base address register update. It forms a virtual address by adding the contents of address register `as` and the contents of address register `at`. 32 bits (four bytes) are read from the physical address. This data is then written to floating-point register `fr` and the virtual address is written back to address register `as`.

If the Region Translation Option (page 156) or the MMU Option (page 158)is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

```
vAddr ← AR[s] + (AR[t])
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     FR[r] ← mem32
     AR[s] ← vAddr
endif
```

Exceptions

- Memory Load Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

MADD.S Multiply and Add Single

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
MADD.S fr, fs, ft
```

Description

Using IEEE754 single-precision arithmetic, `MADD.S` multiplies the contents of floating-point registers `fs` and `ft`, adds the product to the contents of floating-point register `fr`, and then writes the sum back to floating-point register `fr`. The computation is performed with no intermediate round.

Operation

```
FR[r] ← FR[r] $+_s$ (FR[s] $\times_s$ FR[t]) ($\times_s$ does not round)
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Maximum Value MAX

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Miscellaneous Operations Option (See Section 4.3.8 on page 62)

Assembler Syntax

```
MAX ar, as, at
```

Description

`MAX` computes the maximum of the twos complement contents of address registers `as` and `at` and writes the result to address register `ar`.

Operation

```
AR[r] ← if AR[s] < AR[t] then AR[t] else AR[s]
```

Exceptions

- EveryInstR Group (see page 244)

MAXU Maximum Value Unsigned

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 1	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Miscellaneous Operations Option (See Section 4.3.8 on page 62)

Assembler Syntax

```
MAXU ar, as, at
```

Description

`MAXU` computes the maximum of the unsigned contents of address registers `as` and `at` and writes the result to address register `ar`.

Operation

```
AR[r] ← if (0||AR[s]) < (0||AR[t]) then AR[t] else AR[s]
```

Exceptions

- EveryInstR Group (see page 244)

Memory Wait MEMW

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	1 1 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
MEMW
```

Description

MEMW ensures that all previous load, store, acquire, release, prefetch, and cache instructions perform before performing any subsequent load, store, acquire, release, prefetch, or cache instructions. MEMW is intended to implement the volatile attribute of languages such as C and C++. The compiler should separate all volatile loads and stores with a MEMW instruction. ISYNC should be used to cause instruction fetches to wait as MEMW will have no effect on them.

On processor/system implementations that always reference memory in program order, MEMW may be a no-op. Implementations that reorder load, store, or cache instructions, or which perform merging of stores (for example, in a write buffer) must order such memory references so that all memory references executed before MEMW are performed before any memory references that are executed after MEMW.

Because the instruction execution pipeline is implementation-specific, the operation section below specifies only a call to the implementation's memw function.

Operation

```
memw()
```

Exceptions

- EveryInst Group (see page 244)

MIN Minimum Value

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Miscellaneous Operations Option (See Section 4.3.8 on page 62)

Assembler Syntax

```
MIN ar, as, at
```

Description

`MIN` computes the minimum of the twos complement contents of address registers `as` and `at` and writes the result to address register `ar`.

Operation

```
AR[r] ← if AR[s] < AR[t] then AR[s] else AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

Minimum Value Unsigned MINU

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 0	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Miscellaneous Operations Option (See Section 4.3.8 on page 62)

Assembler Syntax

```
MINU ar, as, at
```

Description

`MINU` computes the minimum of the unsigned contents of address registers `as` and `at`, and writes the result to address register `ar`.

Operation

```
AR[r] ← if (0||AR[s]) < (0||AR[t]) then AR[s] else AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

MOV Move

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 0	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Assembler Macro

Assembler Syntax

```
MOV ar, as
```

Description

MOV is an assembler macro that uses the OR instruction (page 466) to move the contents of address register as to address register ar. The assembler input

```
        MOV  ar, as
```

expands into

```
        OR   ar, as, as
```

ar and as should not specify the same register due to the MOV.N restriction.

Assembler Note

The assembler may convert MOV instructions to MOV.N when the Code Density Option is enabled. Prefixing the MOV instruction with an underscore (_MOV) disables this optimization and forces the assembler to generate the OR form of the instruction.

Operation

```
AR[r] ← AR[s]
```

Exceptions

- EveryInstR Group (see page 244)

Narrow Move MOV.N

Instruction Word (RRRN)

15 12	11 8	7 4	3 0
0 0 0 0	s	t	1 1 0 1
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53)

Assembler Syntax

```
MOV.N at, as
```

Description

MOV.N is similar in function to the assembler macro MOV, but has a 16-bit encoding. MOV.N moves the contents of address register as to address register at.

The operation of the processor when at and as specify the same register is undefined and reserved for future use.

Assembler Note

The assembler may convert MOV.N instructions to MOV. Prefixing the MOV.N instruction with an underscore (_MOV.N) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

```
AR[t] ← AR[s]
```

Exceptions

- EveryInstR Group (see page 244)

MOV.S Move Single

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	1 0 1 0	r	s	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
MOV.S fr, fs
```

Description

`MOV.S` moves the contents of floating-point register `fs` to floating-point register `fr`. The move is non-arithmetic; no floating-point exceptions are raised.

Operation

```
FR[r] ← FR[s]
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Move if Equal to Zero MOVEQZ

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 0	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
MOVEQZ ar, as, at
```

Description

MOVEQZ performs a conditional move if equal to zero. If the contents of address register at are zero, then the processor sets address register ar to the contents of address register as. Otherwise, MOVEQZ performs no operation and leaves address register ar unchanged.

The inverse of MOVEQZ is MOVNEZ.

Operation

```
if AR[t] = $0^{32}$ then
     AR[r] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see page 244)

MOVEQZ.S Move Single if Equal to Zero

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 0	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
MOVEQZ.S fr, fs, at
```

Description

`MOVEQZ.S` is a conditional move between floating-point registers based on the value in an address register. If address register `at` contains zero, the contents of floating-point register `fs` are written to floating-point register `fr`. `MOVEQZ.S` is non-arithmetic; no floating-point exceptions are raised.

The inverse of `MOVEQZ.S` is `MOVNEZ.S`.

Operation

```
if AR[t] = $0^{32}$ then
     FR[r] ← FR[s]
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Move if False MOVF

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 0 0	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
MOVF ar, as, bt
```

Description

`MOVF` moves the contents of address register `as` to address register `ar` if Boolean register `bt` is false. Address register `ar` is left unchanged if Boolean register `bt` is true.

The inverse of `MOVF` is `MOVT`.

Operation

```
if not $BR_t$ then
     AR[r] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see page 244)

MOVF.S Move Single if False

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 0 0	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
MOVF.S fr, fs, bt
```

Description

`MOVF.S` is a conditional move between floating-point registers based on the value in a Boolean register. If Boolean register `bt` contains zero, the contents of floating-point register `fs` are written to floating-point register `fr`. `MOVF.S` is non-arithmetic; no floating-point exceptions are raised.

The inverse of `MOVF.S` is `MOVT.S`.

Operation

```
if not $BR_t$ then
     FR[r] ← FR[s]
endif
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Move if Greater Than or Equal to Zero MOVGEZ

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 1	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
MOVGEZ ar, as, at
```

Description

`MOVGEZ` performs a conditional move if greater than or equal to zero. If the contents of address register `at` are greater than or equal to zero (that is, the most significant bit is clear), then the processor sets address register `ar` to the contents of address register `as`. Otherwise, `MOVGEZ` performs no operation and leaves address register `ar` unchanged.

The inverse of `MOVGEZ` is `MOVLTZ`.

Operation

```
if $AR[t]_{31}$ = 0 then
     AR[r] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see page 244)

MOVGEZ.S   Move Single if Greater Than or Eq Zero

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 1	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
MOVGEZ.S fr, fs, at
```

Description

`MOVGEZ.S` is a conditional move between floating-point registers based on the value in an address register. If the contents of address register `at` is greater than or equal to zero (that is, the most significant bit is clear), the contents of floating-point register `fs` are written to floating-point register `fr`. `MOVGEZ.S` is non-arithmetic; no floating-point exceptions are raised.

The inverse of `MOVGEZ.S` is `MOVLTZ.S`.

Operation

```
if $AR[t]_{31}$ = 0 then
     FR[r] ← FR[s]
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Instruction Word (RRI8)

23 20 19 16	15 12	11 8	7 4	3 0
$\text{imm12b}_{7..0}$	1 0 1 0	$\text{imm12b}_{11..8}$	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
MOVI at, -2048..2047
```

Description

`MOVI` sets address register `at` to a constant in the range `-2048..2047` encoded in the instruction word. The constant is stored in two non-contiguous fields of the instruction word. The processor decodes the constant specification by concatenating the two fields and sign-extending the 12-bit value.

Assembler Note

The assembler will convert `MOVI` instructions into a literal load when given an immediate operand that evaluates to a value outside the range `-2048..2047`. The assembler will convert `MOVI` instructions to `MOVI.N` when the Code Density Option is enabled and the immediate operand falls within the available range. Prefixing the `MOVI` instruction with an underscore (`_MOVI`) disables these features and forces the assembler to generate an error for the first case and the wide form of the instruction for the second case.

Operation

$\mathtt{AR[t]} \leftarrow \mathtt{imm12}_{11}{}^{20} \| \mathtt{imm12}$

Exceptions

- EveryInstR Group (see page 244)

MOVI.N Narrow Move Immediate

Instruction Word (RI7)

15 12	11 8	7 6 4	3 0
$\text{imm7}_{3..0}$	s	0 $\text{imm7}_{6..4}$	1 1 0 0
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53)

Assembler Syntax

```
MOVI.N as, -32..95
```

Description

`MOVI.N` is similar to `MOVI`, but has a 16-bit encoding and supports a smaller range of constant values encoded in the instruction word.

`MOVI.N` sets address register `as` to a constant in the range `-32..95` encoded in the instruction word. The constant is stored in two non-contiguous fields of the instruction word. The range is asymmetric around zero because positive constants are more frequent than negative constants. The processor decodes the constant specification by concatenating the two fields and sign-extending the 7-bit value with the logical and of its two most significant bits.

Assembler Note

The assembler may convert `MOVI.N` instructions to `MOVI`. Prefixing the `MOVI.N` instruction with an underscore (`_MOVI.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

$$\text{AR[s]} \leftarrow (\text{imm7}_6 \text{ and } \text{imm7}_5)^{25} \| \text{imm7}$$

Exceptions

- EveryInstR Group (see page 244)

Move if Less Than Zero MOVLTZ

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 0	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
MOVLTZ ar, as, at
```

Description

MOVLTZ performs a conditional move if less than zero. If the contents of address register at are less than zero (that is, the most significant bit is set), then the processor sets address register ar to the contents of address register as. Otherwise, MOVLTZ performs no operation and leaves address register ar unchanged.

The inverse of MOVLTZ is MOVGEZ.

Operation

```
if $AR[t]_{31} \neq 0$ then
     AR[r] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see page 244)

MOVLTZ.S Move Single if Less Than Zero

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 0	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
MOVLTZ.S fr, fs, at
```

Description

MOVLTZ.S is a conditional move between floating-point registers based on the value in an address register. If the contents of address register at is less than zero (that is, the most significant bit is set), the contents of floating-point register fs are written to floating-point register fr. MOVLTZ.S is non-arithmetic; no floating-point exceptions are raised.

The inverse of MOVLTZ.S is MOVGEZ.S.

Operation

```
if $AR[t]_{31} \ne 0$ then
     $FR[r] \leftarrow FR[s]$
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Move if Not-Equal to Zero MOVNEZ

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 1	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
MOVNEZ ar, as, at
```

Description

`MOVNEZ` performs a conditional move if not equal to zero. If the contents of address register `at` are non-zero, then the processor sets address register `ar` to the contents of address register `as`. Otherwise, `MOVNEZ` performs no operation and leaves address register `ar` unchanged.

The inverse of `MOVNEZ` is `MOVEQZ`.

Operation

```
if AR[t] ≠ $0^{32}$ then
     AR[r] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see page 244)

MOVNEZ.S Move Single if Not Equal to Zero

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 1	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
MOVNEZ.S fr, fs, at
```

Description

`MOVNEZ.S` is a conditional move between floating-point registers based on the value in an address register. If the contents of address register `at` is non-zero, the contents of floating-point register `fs` are written to floating-point register `fr`. `MOVNEZ.S` is non-arithmetic; no floating-point exceptions are raised.

The inverse of `MOVNEZ.S` is `MOVEQZ.S`.

Operation

```
if AR[t] $\neq 0^{32}$ then
    FR[r] $\leftarrow$ FR[s]
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Move to Stack Pointer

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 0 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
MOVSP at, as
```

Description

MOVSP provides an atomic window check and register-to-register move. If the caller’s registers are present in the register file, this instruction simply moves the contents of address register as to address register at. If the caller’s registers are not present, MOVSP raises an Alloca exception.

MOVSP is typically used to perform variable-size stack frame allocation. The Xtensa ABI specifies that the caller’s a0-a3 may be stored just below the callee’s stack pointer. When the stack frame is extended, these values may need to be moved. They can only be moved with interrupts and exceptions disabled. This instruction provides a mechanism to test if they must be moved, and if so, to raise an exception to move the data with interrupts and exceptions disabled. The Xtensa ABI also requires that the caller’s return address be in a0 when MOVSP is executed.

Operation

```
if $\mathrm{WindowStart}_{\mathrm{WindowBase-0011..WindowBase-0001}} = 0^3$ then
    Exception (AllocaCause)
else
    AR[t] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(AllocaCause) if Windowed Register Option

MOVT Move if True

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 0 1	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
MOVT ar, as, bt
```

Description

`MOVT` moves the contents of address register `as` to address register `ar` if Boolean register `bt` is true. Address register `ar` is left unchanged if Boolean register `bt` is false.

The inverse of `MOVT` is `MOVF`.

Operation

```
if $BR_t$ then
     AR[r] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see page 244)

Move Single if True MOVT.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 0 1	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
MOVT.S fr, fs, bt
```

Description

`MOVT.S` is a conditional move between floating-point registers based on the value in a Boolean register. If Boolean register `bt` is set, the contents of floating-point register `fs` are written to floating-point register `fr`. `MOVT.S` is non-arithmetic; no floating-point exceptions are raised.

The inverse of `MOVT.S` is `MOVF.S`.

Operation

```
if $BR_t$ then
     FR[r] ← FR[s]
endif
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

MSUB.S Multiply and Subtract Single

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
MSUB.S fr, fs, ft
```

Description

`MSUB.S` multiplies the contents of floating-point registers `fs` and `ft`, subtracts the product from the contents of floating-point register `fr`, and then writes the difference back to floating-point register `fr`. The computation is performed with no intermediate round.

Operation

```
$FR[r] \leftarrow FR[r] -_s (FR[s] \times_s FR[t])$ ($\times_s$ does not round)
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Signed Multiply MUL.AA.*

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 1	0 1 half	0 0 0 0	s	t	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MUL.AA.* as, at
```

Where * expands as follows:

```
MUL.AA.LL - for (half=0)
MUL.AA.HL - for (half=1)
MUL.AA.LH - for (half=2)
MUL.AA.HH - for (half=3)
```

Description

`MUL.AA.*` performs a two’s complement multiply of half of each of the address registers `as` and `at`, producing a 32-bit result. The result is sign-extended to 40 bits and written to the MAC16 accumulator.

Operation

```
$m1 \leftarrow$ if $\mathrm{half}_0$ then $\mathrm{AR[s]}_{31..16}$ else $\mathrm{AR[s]}_{15..0}$
$m2 \leftarrow$ if $\mathrm{half}_1$ then $\mathrm{AR[t]}_{31..16}$ else $\mathrm{AR[t]}_{15..0}$
ACC $\leftarrow$ $(m1_{15}{}^{24} \| m1) \times (m2_{15}{}^{24} \| m2)$
```

Exceptions

- EveryInstR Group (see page 244)

MUL.AD.* Signed Multiply

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 1	0 1 half	0 0 0 0	s	0 y 0 0	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MUL.AD.* as, my
```

Where * expands as follows:

```
MUL.AD.LL - for (half=0)
MUL.AD.HL - for (half=1)
MUL.AD.LH - for (half=2)
MUL.AD.HH - for (half=3)
```

Description

`MUL.AD.*` performs a two's complement multiply of half of address register `as` and half of MAC16 register `my`, producing a 32-bit result. The result is sign-extended to 40 bits and written to the MAC16 accumulator. The `my` operand can designate either MAC16 register `m2` or `m3`.

Operation

```
$m1 \leftarrow$ if $half_0$ then $AR[s]_{31..16}$ else $AR[s]_{15..0}$
$m2 \leftarrow$ if $half_1$ then $MR[1 \| y]_{31..16}$ else $MR[1 \| y]_{15..0}$
$ACC \leftarrow (m1_{15}^{24} \| m1) \times (m2_{15}^{24} \| m2)$
```

Exceptions

- EveryInstR Group (see page 244)

Signed Multiply MUL.DA.*

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 0	0 1 half	0 x 0 0	0 0 0 0	t	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MUL.DA.* mx, at
```

Where * expands as follows:

```
MUL.DA.LL - for (half=0)
MUL.DA.HL - for (half=1)
MUL.DA.LH - for (half=2)
MUL.DA.HH - for (half=3)
```

Description

`MUL.DA.*` performs a two's complement multiply of half of MAC16 register `mx` and half of address register `at`, producing a 32-bit result. The result is sign-extended to 40 bits and written to the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`.

Operation

```
$m1 \leftarrow$ if $\mathrm{half}_0$ then $\mathrm{MR}[0||x]_{31..16}$ else $\mathrm{MR}[0||x]_{15..0}$
$m2 \leftarrow$ if $\mathrm{half}_1$ then $\mathrm{AR}[t]_{31..16}$ else $\mathrm{AR}[t]_{15..0}$
$\mathrm{ACC} \leftarrow (m1_{15}{}^{24}||m1) \times (m2_{15}{}^{24}||m2)$
```

Exceptions

- EveryInstR Group (see page 244)

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 0	0 1 half	0 x 0 0	0 0 0 0	0 y 0 0	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MUL.DD.* mx, my
```

Where * expands as follows:

```
MUL.DD.LL - for (half=0)
MUL.DD.HL - for (half=1)
MUL.DD.LH - for (half=2)
MUL.DD.HH - for (half=3)
```

Description

`MUL.DD.*` performs a two's complement multiply of half of the MAC16 registers `mx` and `my`, producing a 32-bit result. The result is sign-extended to 40 bits and written to the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`. The `my` operand can designate either MAC16 register `m2` or `m3`.

Operation

```
$m1 \leftarrow \text{if } \text{half}_0 \text{ then } \text{MR}[0||x]_{31..16} \text{ else } \text{MR}[0||x]_{15..0}$
$m2 \leftarrow \text{if } \text{half}_1 \text{ then } \text{MR}[1||y]_{31..16} \text{ else } \text{MR}[1||y]_{15..0}$
$\text{ACC} \leftarrow (m1_{15}{}^{24}||m1) \times (m2_{15}{}^{24}||m2)$
```

Exceptions

- EveryInst Group (see page 244)

Multiply Single MUL.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 0	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
MUL.S fr, fs, ft
```

Description

MUL.S computes the IEEE754 single-precision product of the contents of floating-point registers fs and ft and writes the result to floating-point register fr.

Operation

```
FR[r] ← FR[s] $\times_s$ FR[t]
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

MUL16S Multiply 16-bit Signed

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 0 1	0 0 0 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

16-bit Integer Multiply Option (See Section 4.3.4 on page 57)

Assembler Syntax

```
MUL16S ar, as, at
```

Description

`MUL16S` performs a two’s complement multiplication of the least-significant 16 bits of the contents of address registers `as` and `at` and writes the 32-bit product to address register `ar`.

Operation

$$\texttt{AR[r]} \leftarrow (\texttt{AR[s]}_{15}{}^{16} \| \texttt{AR[s]}_{15..0}) \times (\texttt{AR[t]}_{15}{}^{16} \| \texttt{AR[t]}_{15..0})$$

Exceptions

- EveryInstR Group (see page 244)

Multiply 16-bit Unsigned MUL16U

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 0 0	0 0 0 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

16-bit Integer Multiply Option (See Section 4.3.4 on page 57)

Assembler Syntax

```
MUL16U ar, as, at
```

Description

`MUL16U` performs an unsigned multiplication of the least-significant 16 bits of the contents of address registers `as` and `at` and writes the 32-bit product to address register `ar`.

Operation

```
$AR[r] \leftarrow (0^{16} \| AR[s]_{15..0}) \times (0^{16} \| AR[t]_{15..0})$
```

Exceptions

- EveryInstR Group (see page 244)

MULA.AA.* Signed Multiply/Accumulate

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 1	1 0 half	0 0 0 0	s	t	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MULA.AA.* as, at
```

Where * expands as follows:

```
MULA.AA.LL - for (half=0)
MULA.AA.HL - for (half=1)
MULA.AA.LH - for (half=2)
MULA.AA.HH - for (half=3)
```

Description

`MULA.AA.*` performs a two's complement multiply of half of each of the address registers `as` and `at`, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator.

Operation

```
$m1 \leftarrow$ if $half_0$ then $AR[s]_{31..16}$ else $AR[s]_{15..0}$
$m2 \leftarrow$ if $half_1$ then $AR[t]_{31..16}$ else $AR[t]_{15..0}$
$ACC \leftarrow ACC + (m1_{15}{}^{24} || m1) \times (m2_{15}{}^{24} || m2)$
```

Exceptions

- EveryInstR Group (see page 244)

Signed Multiply/Accumulate MULA.AD.*

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 1	1 0 half	0 0 0 0	s	0 y 0 0	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MULA.AD.* as, my
```

Where * expands as follows:

```
MULA.AD.LL - for (half=0)
MULA.AD.HL - for (half=1)
MULA.AD.LH - for (half=2)
MULA.AD.HH - for (half=3)
```

Description

`MULA.AD.*` performs a two's complement multiply of half of address register `as` and half of MAC16 register `my`, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The `my` operand can designate either MAC16 register `m2` or `m3`.

Operation

```
$m1 \leftarrow \text{if } \mathrm{half}_0 \text{ then } \mathrm{AR[s]}_{31..16} \text{ else } \mathrm{AR[s]}_{15..0}$
$m2 \leftarrow \text{if } \mathrm{half}_1 \text{ then } \mathrm{MR}[1 \| y]_{31..16} \text{ else } \mathrm{MR}[1 \| y]_{15..0}$
$\mathrm{ACC} \leftarrow \mathrm{ACC} + ({m1_{15}}^{24} \| m1) \times ({m2_{15}}^{24} \| m2)$
```

Exceptions

- EveryInstR Group (see page 244)

MULA.DA.* Signed Multiply/Accumulate

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 0	1 0 half	0 x 0 0	0 0 0 0	t	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MULA.DA.* mx, at
```

Where * expands as follows:

```
MULA.DA.LL - for (half=0)
MULA.DA.HL - for (half=1)
MULA.DA.LH - for (half=2)
MULA.DA.HH - for (half=3)
```

Description

`MULA.DA.*` performs a two's complement multiply of half of MAC16 register `mx` and half of address register `at`, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`.

Operation

```
$m1 \leftarrow \text{if } \mathrm{half}_0 \text{ then } \mathrm{MR}[0 \| x]_{31..16} \text{ else } \mathrm{MR}[0 \| x]_{15..0}$
$m2 \leftarrow \text{if } \mathrm{half}_1 \text{ then } \mathrm{AR}[t]_{31..16} \text{ else } \mathrm{AR}[t]_{15..0}$
$\mathrm{ACC} \leftarrow \mathrm{ACC} + (m1_{15}{}^{24} \| m1) \times (m2_{15}{}^{24} \| m2)$
```

Exceptions

- EveryInstR Group (see page 244)

Signed Mult/Accum, Ld/Autodec MULA.DA.*.LDDEC

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	1 0 half	0 x w	s	t	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MULA.DA.*.LDDEC mw, as, mx, at
```

Where * expands as follows:

```
MULA.DA.LL.LDDEC - for (half=0)
MULA.DA.HL.LDDEC - for (half=1)
MULA.DA.LH.LDDEC - for (half=2)
MULA.DA.HH.LDDEC - for (half=3)
```

Description

MULA.DA.*.LDDEC performs a parallel load and multiply/accumulate.

First, it performs a two's complement multiply of half of MAC16 register mx and half of address register at, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The mx operand can designate either MAC16 register m0 or m1.

Next, it loads MAC16 register mw from memory using auto-decrement addressing. It forms a virtual address by subtracting 4 from the contents of address register as. Thirty-two bits (four bytes) are read from the physical address. This data is then written to MAC16 register mw, and the virtual address is written back to address register as. The mw operand can designate any of the four MAC16 registers.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

MULA.DA.*.LDDEC Signed Mult/Accum, Ld/Autodec

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

The MAC16 register source `mx` and the MAC16 register destination `mw` may be the same. In this case, the instruction uses the contents of `mx` as the source operand prior to loading `mx` with the load data.

Operation

```
vAddr ← AR[s] − 4
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     m1 ← if $half_0$ then $MR[0||x]_{31..16}$ else $MR[0||x]_{15..0}$
     m2 ← if $half_1$ then $AR[t]_{31..16}$ else $AR[t]_{15..0}$
     ACC ← ACC + $(m1_{15}^{24}||m1) \times (m2_{15}^{24}||m2)$
     AR[s] ← vAddr
     MR[w] ← mem32
endif
```

Exceptions

- Memory Load Group (see page 244)

Signed Mult/Accum, Ld/Autoinc MULA.DA.*.LDINC

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	1 0 half	0 x w	s	t	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MULA.DA.*.LDINC mw, as, mx, at
```

Where * expands as follows:

```
MULA.DA.LL.LDINC - for (half=0)
MULA.DA.HL.LDINC - for (half=1)
MULA.DA.LH.LDINC - for (half=2)
MULA.DA.HH.LDINC - for (half=3)
```

Description

`MULA.DA.*.LDINC` performs a parallel load and multiply/accumulate.

First, it performs a two's complement multiply of half of MAC16 register `mx` and half of address register `at`, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`.

Next, it loads MAC16 register `mw` from memory using auto-increment addressing. It forms a virtual address by adding 4 to the contents of address register `as`. 32 bits (four bytes) are read from the physical address. This data is then written to MAC16 register `mw`, and the virtual address is written back to address register `as`. The `mw` operand can designate any of the four MAC16 registers.

If the Region Translation Option (page 156) or the MMU Option (page 158)is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

MULA.DA.*.LDINC Signed Mult/Accum, Ld/Autoinc

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

The MAC16 register source `mx` and the MAC16 register destination `mw` may be the same. In this case, the instruction uses the contents of `mx` as the source operand prior to loading `mx` with the load data.

Operation

```
vAddr ← AR[s] + 4
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     m1 ← if $half_0$ then $MR[0\|x]_{31..16}$ else $MR[0\|x]_{15..0}$
     m2 ← if $half_1$ then $AR[t]_{31..16}$ else $AR[t]_{15..0}$
     ACC ← ACC + $({m1_{15}}^{24}\|m1) \times ({m2_{15}}^{24}\|m2)$
     AR[s] ← vAddr
     MR[w] ← mem32
endif
```

Exceptions

- Memory Load Group (see page 244)

Signed Multiply/Accumulate MULA.DD.*

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 0	1 0 half	0 x 0 0	0 0 0 0	0 y 0 0	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MULA.DD.* mx, my
```

Where * expands as follows:

```
MULA.DD.LL - for (half=0)
MULA.DD.HL - for (half=1)
MULA.DD.LH - for (half=2)
MULA.DD.HH - for (half=3)
```

Description

MULA.DD.* performs a two's complement multiply of half of each of the MAC16 registers mx and my, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The mx operand can designate either MAC16 register m0 or m1. The my operand can designate either MAC16 register m2 or m3.

Operation

```
$m1 \leftarrow$ if $half_0$ then $MR[0||x]_{31..16}$ else $MR[0||x]_{15..0}$
$m2 \leftarrow$ if $half_1$ then $MR[1||y]_{31..16}$ else $MR[1||y]_{15..0}$
$ACC \leftarrow ACC + ({m1_{15}}^{24}||m1) \times ({m2_{15}}^{24}||m2)$
```

Exceptions

- EveryInst Group (see page 244)

MULA.DD.*.LDDEC Signed Mult/Accum, Ld/Autodec

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 1	1 0 half	0 x w	s	0 y 0 0	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MULA.DD.*.LDDEC mw, as, mx, my
```

Where * expands as follows:

```
MULA.DD.LL.LDDEC - for (half=0)
MULA.DD.HL.LDDEC - for (half=1)
MULA.DD.LH.LDDEC - for (half=2)
MULA.DD.HH.LDDEC - for (half=3)
```

Description

`MULA.DD.*.LDDEC` performs a parallel load and multiply/accumulate.

First, it performs a two's complement multiply of half of the MAC16 registers `mx` and `my`, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`. The `my` operand can designate either MAC16 register `m2` or `m3`.

Next, it loads MAC16 register `mw` from memory using auto-decrement addressing. It forms a virtual address by subtracting 4 from the contents of address register `as`. Thirty-two bits (four bytes) are read from the physical address. This data is then written to MAC16 register `mw`, and the virtual address is written back to address register `as`. The `mw` operand can designate any of the four MAC16 registers.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Signed Mult/Accum, Ld/Autodec MULA.DD.*.LDDEC

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

The MAC16 register destination `mw` may be the same as either MAC16 register source `mx` or `my`. In this case, the instruction uses the contents of `mx` and `my` as the source operands prior to loading `mw` with the load data.

Operation

```
vAddr ← AR[s] − 4
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     m1 ← if $\mathrm{half}_0$ then $\mathrm{MR}[0\|x]_{31..16}$ else $\mathrm{MR}[0\|x]_{15..0}$
     m2 ← if $\mathrm{half}_1$ then $\mathrm{MR}[1\|y]_{31..16}$ else $\mathrm{MR}[1\|y]_{15..0}$
     ACC ← ACC + $({m1_{15}}^{24}\|m1) \times ({m2_{15}}^{24}\|m2)$
     AR[s] ← vAddr
     MR[w] ← mem32
endif
```

Exceptions

- Memory Load Group (see page 244)

MULA.DD.*.LDINC Signed Mult/Accum, Ld/Autoinc

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	1 0 half	0 x w	s	0 y 0 0	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
    MULA.DD.*.LDINC mw, as, mx, my
```

Where * expands as follows:

```
        MULA.DD.LL.LDINC - for (half=0)
        MULA.DD.HL.LDINC - for (half=1)
        MULA.DD.LH.LDINC - for (half=2)
        MULA.DD.HH.LDINC - for (half=3)
```

Description

`MULA.DD.*.LDINC` performs a parallel load and multiply/accumulate.

First, it performs a two’s complement multiply of half of each of the MAC16 registers `mx` and `my`, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`. The `my` operand can designate either MAC16 register `m2` or `m3`.

Next, it loads MAC16 register `mw` from memory using auto-increment addressing. It forms a virtual address by adding 4 to the contents of address register `as`. Thirty-two bits (four bytes) are read from the physical address. This data is then written to MAC16 register `mw`, and the virtual address is written back to address register `as`. The `mw` operand can designate any of the four MAC16 registers.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Signed Mult/Accum, Ld/Autoinc MULA.DD.*.LDINC

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

The MAC16 register destination `mw` may be the same as either MAC16 register source `mx` or `my`. In this case, the instruction uses the contents of `mx` and `my` as the source operands prior to loading `mw` with the load data.

Operation

```
vAddr ← AR[s] + 4
(mem32, error) ← Load32(vAddr)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreErrorCause)
else
     m1 ← if $half_0$ then $MR[0\|x]_{31..16}$ else $MR[0\|x]_{15..0}$
     m2 ← if $half_1$ then $MR[1\|y]_{31..16}$ else $MR[1\|y]_{15..0}$
     ACC ← ACC + $(m1_{15}^{24}\|m1) \times (m2_{15}^{24}\|m2)$
     AR[s] ← vAddr
     MR[w] ← mem32
endif
```

Exceptions

- Memory Load Group (see page 244)

MULL Multiply Low

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 0	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

32-bit Integer Multiply Option (See Section 4.3.5 on page 58)

Assembler Syntax

```
MULL ar, as, at
```

Description

`MULL` performs a 32-bit multiplication of the contents of address registers `as` and `at`, and writes the least significant 32 bits of the product to address register `ar`. Because the least significant product bits are unaffected by the multiplicand and multiplier sign, MULL is useful for both signed and unsigned multiplication.

Operation

```
AR[r] ← AR[s] × AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

Signed Multiply/Subtract MULS.AA.*

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 1	1 1 half	0 0 0 0	s	t	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MULS.AA.* as, at
```

Where * expands as follows:

```
MULS.AA.LL - for (half=0)
MULS.AA.HL - for (half=1)
MULS.AA.LH - for (half=2)
MULS.AA.HH - for (half=3)
```

Description

`MULS.AA.*` performs a two’s complement multiply of half of each of the address registers `as` and `at`, producing a 32-bit result. The result is sign-extended to 40 bits and subtracted from the contents of the MAC16 accumulator.

Operation

```
$m1 \leftarrow$ if $half_0$ then $AR[s]_{31..16}$ else $AR[s]_{15..0}$
$m2 \leftarrow$ if $half_1$ then $AR[t]_{31..16}$ else $AR[t]_{15..0}$
$ACC \leftarrow ACC - (m1_{15}{}^{24} \| m1) \times (m2_{15}{}^{24} \| m2)$
```

Exceptions

- EveryInstR Group (see page 244)

MULS.AD.* Signed Multiply/Subtract

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 1	1 1 half	0 0 0 0	s	0 y 0 0	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MULS.AD.* as, my
```

Where * expands as follows:

```
MULS.AD.LL - for (half=0)
MULS.AD.HL - for (half=1)
MULS.AD.LH - for (half=2)
MULS.AD.HH - for (half=3)
```

Description

`MULS.AD.*` performs a two's complement multiply of half of address register `as` and half of MAC16 register `my`, producing a 32-bit result. The result is sign-extended to 40 bits and subtracted from the contents of the MAC16 accumulator. The `my` operand can designate either MAC16 register `m2` or `m3`.

Operation

```
$m1 \leftarrow$ if $\mathrm{half}_0$ then $\mathrm{AR[s]}_{31..16}$ else $\mathrm{AR[s]}_{15..0}$
$m2 \leftarrow$ if $\mathrm{half}_1$ then $\mathrm{MR[1||y]}_{31..16}$ else $\mathrm{MR[1||y]}_{15..0}$
$\mathrm{ACC} \leftarrow \mathrm{ACC} - (m1_{15}{}^{24}||m1) \times (m2_{15}{}^{24}||m2)$
```

Exceptions

- EveryInstR Group (see page 244)

Signed Multiply/Subtract MULS.DA.*

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 0	1 1 half	0 x 0 0	0 0 0 0	t	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MULS.DA.* mx, at
```

Where * expands as follows:

```
MULS.DA.LL - for (half=0)
MULS.DA.HL - for (half=1)
MULS.DA.LH - for (half=2)
MULS.DA.HH - for (half=3)
```

Description

`MULS.DA.*` performs a two's complement multiply of half of MAC16 register `mx` and half of address register `at`, producing a 32-bit result. The result is sign-extended to 40 bits and subtracted from the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`.

Operation

```
$m1 \leftarrow$ if $\text{half}_0$ then $\text{MR}[0\|\text{x}]_{31..16}$ else $\text{MR}[0\|\text{x}]_{15..0}$
$m2 \leftarrow$ if $\text{half}_1$ then $\text{AR}[\text{t}]_{31..16}$ else $\text{AR}[\text{t}]_{15..0}$
$\text{ACC} \leftarrow \text{ACC} - (m1_{15}{}^{24}\|m1) \times (m2_{15}{}^{24}\|m2)$
```

Exceptions

- EveryInstR Group (see page 244)

MULS.DD.* Signed Multiply/Subtract

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 0	1 1 half	0 x 0 0	0 0 0 0	0 y 0 0	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
MULS.DD.* mx, my
```

Where * expands as follows:

```
MULS.DD.LL - for (half=0)
MULS.DD.HL - for (half=1)
MULS.DD.LH - for (half=2)
MULS.DD.HH - for (half=3)
```

Description

`MULS.DD.*` performs a two's complement multiply of half of each of MAC16 registers `mx` and `my`, producing a 32-bit result. The result is sign-extended to 40 bits and subtracted from the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`. The `my` operand can designate either MAC16 register `m2` or `m3`.

Operation

```
$m1 \leftarrow$ if $\text{half}_0$ then $\text{MR}[0 \| \text{x}]_{31..16}$ else $\text{MR}[0 \| \text{x}]_{15..0}$
$m2 \leftarrow$ if $\text{half}_1$ then $\text{MR}[1 \| \text{y}]_{31..16}$ else $\text{MR}[1 \| \text{y}]_{15..0}$
$\text{ACC} \leftarrow \text{ACC} - ({m1_{15}}^{24} \| m1) \times ({m2_{15}}^{24} \| m2)$
```

Exceptions

- EveryInst Group (see page 244)

Multiply Signed High MULSH

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 1	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

32-bit Integer Multiply Option (See Section 4.3.5 on page 58)

Assembler Syntax

```
MULSH ar, as, at
```

Description

`MULSH` performs a 32-bit two’s complement multiplication of the contents of address registers `as` and `at` and writes the most significant 32 bits of the product to address register `ar`.

Operation

```
$tp \leftarrow (\mathtt{AR[s]}_{31}{}^{32} \| \mathtt{AR[s]}) \times (\mathtt{AR[t]}_{31}{}^{32} \| \mathtt{AR[t]})$
$\mathtt{AR[r]} \leftarrow tp_{63..32}$
```

Exceptions

- EveryInstR Group (see page 244)

MULUH Multiply Unsigned High

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 0	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

32-bit Integer Multiply Option (See Section 4.3.5 on page 58)

Assembler Syntax

```
MULUH ar, as, at
```

Description

`MULUH` performs an unsigned multiplication of the contents of address registers `as` and `at`, and writes the most significant 32 bits of the product to address register `ar`.

Operation

```
$tp \leftarrow (0^{32}\|\mathtt{AR[s]}) \times (0^{32}\|\mathtt{AR[t]})$
$\mathtt{AR[r]} \leftarrow tp_{63..32}$
```

Exceptions

- EveryInstR Group (see page 244)

Negate NEG

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 0	0 0 0 0	r	0 0 0 0	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
NEG ar, at
```

Description

NEG calculates the two’s complement negation of the contents of address register at and writes it to address register ar. Arithmetic overflow is not detected.

Operation

```
AR[r] ← 0 − AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

NEG.S

Negate Single

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	1 0 1 0	r	s	0 1 1 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
NEG.S fr, fs
```

Description

`NEG.S` negates the single-precision value of the contents of floating-point register `fs` and writes the result to floating-point register `fr`.

Operation

```
FR[r] ← $-_s$ FR[s]
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

No-Operation NOP

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	1 1 1 1	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
NOP
```

Description

This instruction performs no operation. It is typically used for instruction alignment. NOP is a 24-bit instruction. For a 16-bit version, see NOP.N.

Assembler Note

The assembler may convert NOP instructions to NOP.N when the Code Density Option is enabled. Prefixing the NOP instruction with an underscore (_NOP) disables this optimization and forces the assembler to generate the wide form of the instruction.

Operation

```
none
```

Exceptions

- EveryInst Group (see page 244)

Implementation Notes

In some implementations NOP is not an instruction but only an assembler macro that uses the instruction “OR An, An, An” (with An a convenient register).

NOP.N

Narrow No-Operation

Instruction Word (RRRN)

15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 0	0 0 1 1	1 1 0 1
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53)

Assembler Syntax

```
NOP.N
```

Description

This instruction performs no operation. It is typically used for instruction alignment. `NOP.N` is a 16-bit instruction. For a 24-bit version, see `NOP`.

Assembler Note

The assembler may convert `NOP.N` instructions to `NOP`. Prefixing the `NOP.N` instruction with an underscore (`_NOP.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

```
none
```

Exceptions

- EveryInst Group (see page 244)

Normalization Shift Amount

NSA

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 0 0	1 1 1 0	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Miscellaneous Operations Option (See Section 4.3.8 on page 62)

Assembler Syntax

```
NSA at, as
```

Description

`NSA` calculates the left shift amount that will normalize the twos complement contents of address register `as` and writes this amount (in the range 0 to 31) to address register `at`. If `as` contains 0 or -1, `NSA` returns 31. Using `SSL` and `SLL` to shift `as` left by the `NSA` result yields the smallest value for which bits 31 and 30 differ unless `as` contains 0.

Operation

```
$sign \leftarrow AR[s]_{31}$
if $AR[s]_{30..0} = sign^{31}$ then
     $AR[t] \leftarrow 31$
else
     $b4 \leftarrow AR[s]_{30..16} = sign^{15}$
     $t3 \leftarrow$ if $b4$ then $AR[s]_{15..0}$ else $AR[s]_{31..16}$
     $b3 \leftarrow t3_{15..8} = sign^{8}$
     $t2 \leftarrow$ if $b3$ then $t3_{7..0}$ else $t3_{15..8}$
     $b2 \leftarrow t3_{7..4} = sign^{4}$
     $t1 \leftarrow$ if $b2$ then $t2_{3..0}$ else $t2_{7..4}$
     $b1 \leftarrow t3_{3..2} = sign^{2}$
     $b0 \leftarrow$ if $b1$ then $t1_{1} = sign$ else $t1_{3} = sign$
     $AR[t] \leftarrow 0^{27} \| ((b4 \| b3 \| b2 \| b1 \| b0) - 1)$
endif
```

Exceptions

- EveryInstR Group (see page 244)

NSAU Normalization Shift Amount Unsigned

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 0 0	1 1 1 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Miscellaneous Operations Option (See Section 4.3.8 on page 62)

Assembler Syntax

```
NSAU at, as
```

Description

`NSAU` calculates the left shift amount that will normalize the unsigned contents of address register `as` and writes this amount (in the range 0 to 32) to address register `at`. If `as` contains 0, `NSAU` returns 32. Using `SSL` and `SLL` to shift `as` left by the `NSAU` result yields the smallest value for which bit 31 is set, unless `as` contains 0.

Operation

```
if $AR[s] = 0^{32}$ then
     $AR[t] \leftarrow 32$
else
     $b4 \leftarrow AR[s]_{31..16} = 0^{16}$
     $t3 \leftarrow$ if $b4$ then $AR[s]_{15..0}$ else $AR[s]_{31..16}$
     $b3 \leftarrow t3_{15..8} = 0^{8}$
     $t2 \leftarrow$ if $b3$ then $t3_{7..0}$ else $t3_{15..8}$
     $b2 \leftarrow t2_{7..4} = 0^{4}$
     $t1 \leftarrow$ if $b2$ then $t2_{3..0}$ else $t2_{7..4}$
     $b1 \leftarrow t1_{3..2} = 0^{2}$
     $b0 \leftarrow$ if $b1$ then $t1_{1} = 0$ else $t1_{3} = 0$
     $AR[t] \leftarrow 0^{27} || b4 || b3 || b2 || b1 || b0$
endif
```

Exceptions

- EveryInstR Group (see page 244)

Compare Single Equal

OEQ.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 0	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
OEQ.S br, fs, ft
```

Description

`OEQ.S` compares the contents of floating-point registers `fs` and `ft` for IEEE754 equality. If the values are ordered and equal then Boolean register `br` is set to 1, otherwise `br` is set to 0. IEEE754 specifies that +0 and –0 compare as equal. IEEE754 floating-point values are ordered if neither is a NaN.

Operation

```
$BR_r$ ← not isNaN(FR[s]) and not isNaN(FR[t])
              and (FR[s] $=_s$ FR[t])
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

OLE.S Compare Single Ord & Less Than or Equal

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 0	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
OLE.S br, fs, ft
```

Description

OLE.S compares the contents of floating-point registers fs and ft. If the contents of fs are ordered with, and less than or equal to the contents of ft, then Boolean register br is set to 1, otherwise br is set to 0. According to IEEE754, +0 and −0 compare as equal. IEEE754 floating-point values are ordered if neither is a NaN.

Operation

```
$BR_r$ ← not isNaN(FR[s]) and not isNaN(FR[t])
          and (FR[s] $\leq_s$ FR[t])
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Compare Single Ordered and Less Than OLT.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
OLT.S br, fs, ft
```

Description

`OLT.S` compares the contents of floating-point registers `fs` and `ft`. If the contents of `fs` are ordered with and less than the contents of `ft` then Boolean register `br` is set to 1, otherwise br is set to 0. According to IEEE754, +0 and −0 compare as equal. IEEE754 floating-point values are ordered if neither is a NaN.

Operation

```
$BR_r$ ← not isNaN(FR[s]) and not isNaN(FR[t])
            and (FR[s] $<_s$ FR[t])
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 0	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
OR ar, as, at
```

Description

OR calculates the bitwise logical or of address registers as and at. The result is written to address register ar.

Operation

```
AR[r] ← AR[s] or AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

Boolean Or ORB

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 0	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
ORB br, bs, bt
```

Description

ORB performs the logical or of Boolean registers bs and bt, and writes the result to Boolean register br.

When the sense of one of the source Booleans is inverted (0 → true, 1 → false), use ORBC. When the sense of both of the source Booleans is inverted, use ANDB and an inverted test of the result.

Operation

```
$BR_r \leftarrow BR_s$ or $BR_t$
```

Exceptions

- EveryInst Group (see page 244)

ORBC Boolean Or with Complement

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 1	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
ORBC br, bs, bt
```

Description

ORBC performs the logical or of Boolean register bs with the logical complement of Boolean register bt and writes the result to Boolean register br.

Operation

```
$BR_r \leftarrow BR_s$ or not $BR_t$
```

Exceptions

- EveryInst Group (see page 244)

Probe Data TLB PDTLB

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	0 0 0 0	1 1 0 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Region Translation Option (page 156) or the MMU Option (page 158)

Assembler Syntax

```
PDTLB at, as
```

Description

PDTLB searches the data TLB for an entry that translates the virtual address in address register as and writes the way and index of that entry to address register at. If no entry matches, zero is written to the hit bit of at. The value written to at is implementation-specific, but in all implementations a value with the hit bit set is suitable as an input to the IDTLB or WDTLB instructions. See Section 4.6 on page 138 for information on the result register formats for specific memory protection and translation options.

PDTLB is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     (match, vpn, ei, wi) ← ProbeDataTLB(AR[s])
     if match > 1 then
          EXCVADDR ← AR[s]
          Exception (LoadStoreTLBMultiHit)
     else
          AR[t] ← PackDataTLBEntrySpec(match, vpn, ei, wi)
     endif
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(LoadStoreTLBMultiHitCause) if Region Protection Option or MMU Option
- GenExcep(PrivilegedCause) if Exception Option

PITLB Probe Instruction TLB

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	0 0 0 0	0 1 0 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Region Translation Option (page 156) or the MMU Option (page 158)

Assembler Syntax

```
PITLB at, as
```

Description

PITLB searches the Instruction TLB for an entry that translates the virtual address in address register as and writes the way and index of that entry to address register at. If no entry matches, zero is written to the hit bit of at. The value written to at is implementation-specific, but in all implementations a value with the hit bit set is suitable as an input to the IITLB or WITLB instructions. See Section 4.6 on page 138 for information on the result register formats for specific memory protection and translation options.

PITLB is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     (match, vpn, ei, wi) ← ProbeInstTLB(AR[s])
     if match > 1 then
          EXCVADDR ← AR[s]
          Exception (InstructionFetchTLBMultiHit)
     else
          AR[t] ← PackInstTLBEntrySpec(match, vpn, ei, wi)
     endif
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Quotient Signed QUOS

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 0 1	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

32-bit Integer Divide Option (See Section 4.3.6 on page 59)

Assembler Syntax

```
QUOS ar, as, at
```

Description

QUOS performs a 32-bit two's complement division of the contents of address register as by the contents of address register at and writes the quotient to address register ar. The ambiguity which exists when either address register as or address register at is negative is resolved by requiring the product of the quotient and address register at to be smaller in absolute value than the address register as. If the contents of address register at are zero, QUOS raises an Integer Divide by Zero exception instead of writing a result. Overflow (-2147483648 divided by -1) is not detected.

Operation

```
if AR[t] = $0^{32}$ then
     Exception (IntegerDivideByZero)
else
     AR[r] ← AR[s] quo AR[t]
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(IntegerDivideByZeroCause) if 32-bit Integer Divide Option

QUOU Quotient Unsigned

Instruction Word (RRR)

23 … 20	19 … 16	15 … 12	11 … 8	7 … 4	3 … 0
1 1 0 0	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

32-bit Integer Divide Option (See Section 4.3.6 on page 59)

Assembler Syntax

```
QUOU ar, as, at
```

Description

`QUOU` performs a 32-bit unsigned division of the contents of address register `as` by the contents of address register `at` and writes the quotient to address register `ar`. If the contents of address register `at` are zero, `QUOU` raises an Integer Divide by Zero exception instead of writing a result.

Operation

```
if AR[t] = $0^{32}$ then
     Exception (IntegerDivideByZero)
else
     $tq$ ← (0||AR[s]) quo (0||AR[t])
     AR[r] ← $tq_{31..0}$
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(IntegerDivideByZeroCause) if 32-bit Integer Divide Option

Read Data TLB Entry Virtual RDTLB0

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	0 0 0 0	1 0 1 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Region Translation Option (page 156) or the MMU Option (page 158)

Assembler Syntax

```
RDTLB0 at, as
```

Description

`RDTLB0` reads the data TLB entry specified by the contents of address register `as` and writes the Virtual Page Number (VPN) and address space ID (ASID) to address register `at`. See Section 4.6 on page 138 for information on the address and result register formats for specific memory protection and translation options.

`RDTLB0` is a privileged instruction.

Operation

```
AR[t] ← RDTLB0(AR[s])
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

RDTLB1 Read Data TLB Entry Translation

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	0 0 0 0	1 1 1 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Region Translation Option (page 156) or the MMU Option (page 158)

Assembler Syntax

```
RDTLB1 at, as
```

Description

`RDTLB1` reads the data TLB entry specified by the contents of address register `as` and writes the Physical Page Number (PPN) and cache attribute (CA) to address register `at`. See Section 4.6 on page 138 for information on the address and result register formats for specific memory protection and translation options.

`RDTLB1` is a privileged instruction.

Operation

```
AR[t] ← RDTLB1(AR[s])
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Remainder Signed REMS

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

32-bit Integer Divide Option (See Section 4.3.6 on page 59)

Assembler Syntax

```
REMS ar, as, at
```

Description

REMS performs a 32-bit two’s complement division of the contents of address register as by the contents of address register at and writes the remainder to address register ar. The ambiguity which exists when either address register as or address register at is negative is resolved by requiring the remainder to have the same sign as address register as. If the contents of address register at are zero, REMS raises an Integer Divide by Zero exception instead of writing a result.

Operation

```
if AR[t] = $0^{32}$ then
     Exception (IntegerDivideByZero)
else
     AR[r] ← AR[s] rem AR[t]
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(IntegerDivideByZeroCause) if 32-bit Integer Divide Option

REMU Remainder Unsigned

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 0	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

32-bit Integer Divide Option (See Section 4.3.6 on page 59)

Assembler Syntax

```
REMU ar, as, at
```

Description

`REMU` performs a 32-bit unsigned division of the contents of address register `as` by the contents of address register `at` and writes the remainder to address register `ar`. If the contents of address register `at` are zero, `REMU` raises an Integer Divide by Zero exception instead of writing a result.

Operation

```
if AR[t] = $0^{32}$ then
     Exception (IntegerDivideByZero)
else
     $tr$ ← (0||AR[s]) rem (0||AR[t])
     AR[r] ← $tr_{31..0}$
endif
```

Exceptions

- EveryInstR Group (see page 244)

GenExcep(IntegerDivideByZeroCause) if 32-bit Integer Divide Option

Read External Register RER

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 0 0	0 1 1 0	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
RER at, as
```

Description

RER reads one of a set of "External Registers". It is in some ways similar to the RSR.* instruction except that the registers being read are not defined by the Xtensa ISA and are conceptually outside the processor core. They are read through processor ports.

Address register as is used to determine which register is to be read and the result is placed in address register at. When no External Register is addressed by the value in address register as, the result in address register at is undefined. The entire address space is reserved for use by Tensilica. RER and WER are managed by the processor core so that the requests appear on the processor ports in program order. External logic is responsible for extending that order to the registers themselves.

RER is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     Read External Register as defined outside the processor.
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

RET Non-Windowed Return

Instruction Word (CALLX)

23			20	19			16	15			12	11			8	7	6	5	4	3			0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
4				4				4				4				2		2		4			

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
RET
```

Description

RET returns from a routine called by CALL0 or CALLX0. It is equivalent to the instruction

```
        JX    A0
```

RET exists as a separate instruction because some Xtensa ISA implementations may realize performance advantages from treating this operation as a special case.

Assembler Note

The assembler may convert RET instructions to RET.N when the Code Density Option is enabled. Prefixing the RET instruction with an underscore (_RET) disables this optimization and forces the assembler to generate the wide form of the instruction.

Operation

```
nextPC ← AR[0]
```

Exceptions

- EveryInst Group (see page 244)

Narrow Non-Windowed Return RET.N

Instruction Word (RRRN)

15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 0	0 0 0 0	1 1 0 1
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53)

Assembler Syntax

```
RET.N
```

Description

`RET.N` is the same as `RET` in a 16-bit encoding. `RET` returns from a routine called by `CALL0` or `CALLX0`.

Assembler Note

The assembler may convert `RET.N` instructions to `RET`. Prefixing the `RET.N` instruction with an underscore (`_RET.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

```
nextPC ← AR[0]
```

Exceptions

- EveryInst Group (see page 244)

RETW Windowed Return

Instruction Word (CALLX)

23 20	19 16	15 12	11 8	7 6	5 4	3 0
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 0	0 1	0 0 0 0
4	4	4	4	2	2	4

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
    RETW
```

Description

`RETW` returns from a subroutine called by `CALL4`, `CALL8`, `CALL12`, `CALLX4`, `CALLX8`, or `CALLX12`, and that had `ENTRY` as its first instruction.

`RETW` uses bits `29..0` of address register `a0` as the low 30 bits of the return address and bits `31..30` of the address of the `RETW` as the high two bits of the return address. Bits `31..30` of `a0` are used as the caller's window increment.

`RETW` subtracts the window increment from `WindowBase` to return to the caller's registers. It then checks the `WindowStart` bit for this `WindowBase`. If it is set, then the caller's registers still reside in the register file, and `RETW` completes by clearing its own `WindowStart` bit and jumping to the return address. If the `WindowStart` bit is clear, then the caller's registers have been stored into the stack, so `RETW` signals one of window underflow's 4, 8, or 12, based on the size of the caller's window increment. The underflow handler is invoked with `WindowBase` decremented, a minor exception to the rule that instructions aborted by an exception have no side effects to the operating state of the processor. The processor stores the previous value of `WindowBase` in `PS.OWB` so that it can be restored by `RFWU`.

The window underflow handler is expected to restore the caller's registers, set the caller's `WindowStart` bit, and then return (see `RFWU`) to re-execute the `RETW`, which will then complete.

The operation of this instruction is undefined if AR[0]31..30 is `02`, if PS.WOE is `0`, if `PS.EXCM` is `1`, or if the first set bit among [WindowStartWindowBase-1, WindowStartWindowBase-2, WindowStartWindowBase-3] is anything other than WindowStartWindowBase-n, where `n` is `AR[0]`$_{31..30}$. (If none of the three bits is set, an underflow exception will be raised as described above, but if the wrong first one is set, the state is not legal.) Some implementations raise an illegal instruction exception in these cases as a debugging aid.

Assembler Note

The assembler may convert `RETW` instructions to `RETW.N` when the Code Density Option is enabled. Prefixing the `RETW` instruction with an underscore (`_RETW`) disables this optimization and forces the assembler to generate the wide form of the instruction.

Operation

```
n ← $AR[0]_{31..30}$
nextPC ← $PC_{31..30} \| AR[0]_{29..0}$
owb ← WindowBase
m ← if $WindowStart_{WindowBase-4'b0001}$ then 2'b01
 elsif $WindowStart_{WindowBase-4'b0010}$ then 2'b10
 elsif $WindowStart_{WindowBase-4'b0011}$ then 2'b11
 else 2'b00
if n=2'b00 | (m≠2'b00 & m≠n) | PS.WOE=0 | PS.EXCM=1 then
     -- undefined operation
     -- may raise illegal instruction exception
else
     WindowBase ← WindowBase − ($0^2 \| n$)
     if $WindowStart_{WindowBase}$ ≠ 0 then
          $WindowStart_{owb}$ ← 0
     else
          -- Underflow exception
          PS.EXCM ← 1
          EPC[1] ← PC
          PS.OWB ← owb
          nextPC ← if n = 2'b01 then WindowUnderflow4
               else if n = 2'b10 then WindowUnderflow8
               else WindowUnderflow12
     endif
endif
```

Exceptions

- EveryInst Group (see page 244)
- WindowUnderExcep if Windowed Register Option

RETW.N Narrow Windowed Return

Instruction Word (RRRN)

15 … 12	11 … 8	7 … 4	3 … 0
1 1 1 1	0 0 0 0	0 0 0 1	1 1 0 1
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53) and Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
RETW.N
```

Description

`RETW.N` is the same as `RETW` in a 16-bit encoding.

Assembler Note

The assembler may convert `RETW.N` instructions to `RETW`. Prefixing the `RETW.N` instruction with an underscore (`_RETW.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

```
$n \leftarrow \mathrm{AR[0]}_{31..30}$
$\mathrm{nextPC} \leftarrow \mathrm{PC}_{31..30} \| \mathrm{AR[0]}_{29..0}$
$owb \leftarrow \mathrm{WindowBase}$
$m \leftarrow$ if $\mathrm{WindowStart}_{\mathrm{WindowBase-4'b0001}}$ then 2'b01
 elsif $\mathrm{WindowStart}_{\mathrm{WindowBase-4'b0010}}$ then 2'b10
 elsif $\mathrm{WindowStart}_{\mathrm{WindowBase-4'b0011}}$ then 2'b11
 else 2'b00
if $n$=2'b00 | ($m \neq$2'b00 & $m \neq n$) | PS.WOE=0 | PS.EXCM=1 then
     -- undefined operation
     -- may raise illegal instruction exception
else
     $\mathrm{WindowBase} \leftarrow \mathrm{WindowBase} - (0^2 \| n)$
     if $\mathrm{WindowStart}_{\mathrm{WindowBase}} \neq 0$ then
          $\mathrm{WindowStart}_{owb} \leftarrow 0$
     else
          -- Underflow exception
          $\mathrm{PS.EXCM} \leftarrow 1$
          $\mathrm{EPC[1]} \leftarrow \mathrm{PC}$
```

```
            PS.OWB ← owb
            nextPC ← if n = 2'b01 then WindowUnderflow4
                 else if n = 2'b10 then WindowUnderflow8
                 else WindowUnderflow12
        endif
    endif
```

Exceptions

- EveryInst Group (see page 244)
- WindowUnderExcep if Windowed Register Option

RFDD Return from Debug and Dispatch

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 1	1 1 1 0	0 0 0 s_0	0 0 0 1	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Debug Option (See Section 4.7.6 on page 197) and OCD, Implementation-Specific

Assembler Syntax

```
RFDD
```

Description

This instruction is used only in On-Chip Debug Mode and exists only in some implementations. It is an illegal instruction when the processor is not in On-Chip Debug Mode. See the *Tensilica On-Chip Debugging Guide* for a description of its operation.

Exceptions

- EveryInst Group (see page 244)
- GenExcep(IllegalInstructionCause) if Exception Option

Return from Double Exception RFDE

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 1	0 0 1 0	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Exception Option (See Section 4.4.1 on page 82)

Assembler Syntax

```
RFDE
```

Description

RFDE returns from an exception that went to the double exception vector (that is, an exception raised while the processor was executing with PS.EXCM set). It is similar to RFE, but PS.EXCM is not cleared, and DEPC, if it exists, is used instead of EPC[1]. RFDE simply jumps to the exception PC. PS.UM and PS.WOE are left unchanged.

RFDE is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
elsif NDEPC=1 then
     nextPC ¨ DEPC
else
     nextPC ← EPC[1]
endif
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 1	1 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Debug Option (See Section 4.7.6 on page 197) and OCD, Implementation-Specific

Assembler Syntax

```
RFDO
```

Description

This instruction is used only in On-Chip Debug Mode and exists only in some implementations. It is an illegal instruction when the processor is not in On-Chip Debug Mode. See the *Tensilica On-Chip Debugging Guide* for a description of its operation.

Exceptions

- EveryInst Group (see page 244)
- GenExcep(IllegalInstructionCause) if Exception Option

Return from Exception RFE

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Exception Option (See Section 4.4.1 on page 82)

Assembler Syntax

```
RFE
```

Description

`RFE` returns from either the UserExceptionVector or the KernelExceptionVector. `RFE` sets `PS.EXCM` back to `0`, and then jumps to the address in `EPC[1]`. `PS.UM` and `PS.WOE` are left unchanged.

RFE is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     PS.EXCM ← 0
     nextPC ← EPC[1]
endif
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

RFI Return from High-Priority Interrupt

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 1	level	0 0 0 1	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

High-Priority Interrupt Option (See Section 4.4.5 on page 106)

Assembler Syntax

```
RFI 0..15
```

Description

`RFI` returns from a high-priority interrupt. It restores the `PS` from `EPS[level]` and jumps to the address in `EPC[level]`. `Level` is given as a constant `2..(NLEVEL+NNMI)` in the instruction word. The operation of this opcode when level is 0 or 1 or greater than (`NLEVEL+NNMI`) is undefined.

`RFI` is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     nextPC ← EPC[level]
     PS ← EPS[level]
endif
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Return from Memory Error RFME

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 1 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Memory ECC/Parity Option (See Section 4.5.14 on page 128)

Assembler Syntax

```
RFME
```

Description

`RFME` returns from a memory error exception. It restores the `PS` from `MEPS` and jumps to the address in `MEPC`. In addition, the `MEME` bit of the `MESR` register is cleared.

`RFME` is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     nextPC ← MEPC
     PS ← MEPS
     MESR.MEME ← 0
endif
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

RFR Move FR to AR

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	1 0 1 0	r	s	0 1 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
RFR ar, fs
```

Description

`RFR` moves the contents of floating-point register `fs` to address register `ar`. The move is non-arithmetic; no floating-point exceptions are raised.

Operation

```
AR[r] ← FR[s]
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Return from User-Mode Exception RFUE

Instruction Word (RRR)

23			20	19			16	15			12	11			8	7			4	3			0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0
4				4				4				4				4				4			

Required Configuration Option

Exception Option (Xtensa Exception Architecture 1 Only)

Assembler Syntax

```
RFUE
```

Description

`RFUE` exists only in Xtensa Exception Architecture 1 (see Section A.2 “Xtensa Exception Architecture 1” on page 611). It is an illegal instruction in current Xtensa implementations.

Exceptions

- EveryInst Group (see page 244)
- GenExcep(IllegalInstructionCause) if Exception Option

RFWO Return from Window Overflow

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 1	0 1 0 0	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
RFWO
```

Description

RFWO returns from an exception that went to one of the three window overflow vectors. It sets PS.EXCM back to 0, clears the WindowStart bit of the registers that were spilled, restores WindowBase from PS.OWB, and then jumps to the address in EPC[1]. PS.UM is left unchanged.

RFWO is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     PS.EXCM ← 0
     nextPC ← EPC[1]
     $\mathrm{WindowStart}_{\mathrm{WindowBase}}$ ← 0
     WindowBase ← PS.OWB
endif
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Return From Window Underflow RFWU

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 0 1 1	0 1 0 1	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
RFWU
```

Description

RFWU returns from an exception that went to one of the three window underflow vectors. It sets PS.EXCM back to 0, sets the WindowStart bit of the registers that were reloaded, restores WindowBase from PS.OWB, and then jumps to the address in EPC[1]. PS.UM is left unchanged.

RFWU is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     PS.EXCM ← 0
     nextPC ← EPC[1]
     $\text{WindowStart}_{\text{WindowBase}}$ ← 1
     WindowBase ← PS.OWB
endif
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

RITLB0 Read Instruction TLB Entry Virtual

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	0 0 0 0	0 0 1 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Region Translation Option (page 156) or the MMU Option (page 158)

Assembler Syntax

```
RITLB0 at, as
```

Description

RITLB0 reads the instruction TLB entry specified by the contents of address register as and writes the Virtual Page Number (VPN) and address space ID (ASID) to address register at. See Section 4.6 on page 138 for information on the address and result register formats for specific memory protection and translation options.

RITLB0 is a privileged instruction.

Operation

```
AR[t] ← RITLB0(AR[s])
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Read Instruction TLB Entry Translation RITLB1

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	0 0 0 0	0 1 1 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Region Translation Option (page 156) or the MMU Option (page 158)

Assembler Syntax

```
RITLB1 at, as
```

Description

`RITLB1` reads the instruction TLB entry specified by the contents of address register `as` and writes the Physical Page Number (PPN) and cache attribute (CA) to address register `at`. See Section 4.6 on page 138 for information on the address and result register formats for specific memory protection and translation options.

`RITLB1` is a privileged instruction.

Operation

```
AR[t] ← RITLB1(AR[s])
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

ROTW

Rotate Window

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 0 0	1 0 0 0	0 0 0 0	imm4	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
ROTW -8..7
```

Description

ROTW adds a constant to WindowBase, thereby moving the current window into the register file. ROTW is intended for use in exception handlers and context switch code.

ROTW is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     WindowBase ← WindowBase + imm4
endif
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Round Single to Fixed ROUND.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 0	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
ROUND.S ar, fs, 0..15
```

Description

`ROUND.S` converts the contents of floating-point register `fs` from single-precision to signed integer format, rounding toward the nearest. The single-precision value is first scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 2.0, 4.0, …, 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0` and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. For positive overflow (value ≥ `32'h7fffffff`), positive infinity, or NaN, `32'h7fffffff` is returned; for negative overflow (value ≤ `32'h80000000`) or negative infinity, `32'h80000000` is returned. The result is written to address register `ar`.

Operation

```
$AR[r] \leftarrow round_s(FR[s] \times_s pow_s(2.0,t))$
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

RSIL Read and Set Interrupt Level

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 1 1 0	imm4	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Interrupt Option (See Section 4.4.4 on page 100)

Assembler Syntax

```
RSIL at, 0..15
```

Description

RSIL first reads the PS Special Register (described in Table 4–63 on page 87, PS Register Fields), writes this value to address register at, and then sets PS.INTLEVEL to a constant in the range 0..15 encoded in the instruction word. Interrupts at and below the PS.INTLEVEL level are disabled.

A WSR.PS or XSR.PS followed by an RSIL should be separated with an ESYNC to guarantee the value written is read back.

On some Xtensa ISA implementations the latency of RSIL is greater than one cycle, and so it is advantageous to schedule uses of the RSIL result later.

RSIL is typically used as follows:

```
RSIL a2, newlevel
code to be executed at newlevel
WSR.PS    a2
```

The instruction following the RSIL is guaranteed to be executed at the new interrupt level specified in PS.INTLEVEL, therefore it is not necessary to insert one of the SYNC instructions to force the interrupt level change to take effect.

RSIL is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     AR[t] ← PS
```

```
        PS.INTLEVEL ← s
    endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Instruction Word (RSR)

23 20	19 16	15 8	7 4	3 0
0 0 0 0	0 0 1 1	sr	t	0 0 0 0
4	4	8	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
RSR.* at

RSR at, *

RSR at, 0..255
```

Description

RSR.* reads the Special Registers that are described in Section 3.8.10 “Processor Control Instructions” on page 45. See Section 5.3 on page 208 for more detailed information on the operation of this instruction for each Special Register.

The contents of the Special Register designated by the 8-bit sr field of the instruction word are written to address register at. The name of the Special Register is used in place of the ‘*’ in the assembler syntax above and the translation is made to the 8-bit sr field by the assembler.

RSR is an assembler macro for RSR.* that provides compatibility with the older versions of the instruction containing either the name or the number of the Special Register.

A WSR.* followed by an RSR.* to the same register should be separated with ESYNC to guarantee the value written is read back. On some Xtensa ISA implementations, the latency of RSR.* is greater than one cycle, and so it is advantageous to schedule other instructions before instructions that use the RSR.* result.

RSR.* with Special Register numbers ≥ 64 is privileged. An RSR.* for an unconfigured register generally will raise an illegal instruction exception.

Operation

```
sr ← if msbFirst then s||r else r||s
if sr ≥ 64 and CRING ≠ 0 then
```

```
        Exception (PrivilegedInstructionCause)
    else
        see the Tables in Section 5.3 on page 208
    endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(IllegalInstructionCause) if Exception Option
- GenExcep(PrivilegedCause) if Exception Option

RSYNC Register Read Synchronize

Instruction Word (RRR)

23			20	19			16	15			12	11			8	7			4	3			0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
4				4				4				4				4				4			

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
RSYNC
```

Description

`RSYNC` waits for all previously fetched `WSR.*` instructions to be performed before interpreting the register fields of the next instruction. This operation is also performed as part of `ISYNC`. `ESYNC` and `DSYNC` are performed as part of this instruction.

This instruction is appropriate after `WSR.WindowBase`, `WSR.WindowStart`, `WSR.PS`, `WSR.CPENABLE`, or `WSR.EPS*` instructions before using their results. See the Special Register Tables in Section 5.3 on page 208 for a complete description of the uses of the `RSYNC` instruction.

Because the instruction execution pipeline is implementation-specific, the operation section below specifies only a call to the implementation's `rsync` function.

Operation

```
rsync()
```

Exceptions

- EveryInst Group (see page 244)

Read User Register RUR.*

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 0	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

No Option - instructions created from the TIE language (See Section 4.3.9.2 “Coprocessor Context Switch” on page 64)

Assembler Syntax

```
RUR.* ar

RUR ar, *
```

Description

RUR.* reads TIE state that has been grouped into 32-bit quantities by the TIE user_register statement. The name in the user_register statement replaces the “*” in the instruction name and causes the correct register number to be placed in the st field of the encoded instruction. The contents of the TIE user_register designated by the 8-bit number 16*s+t are written to address register ar. Here s and t are the numbers corresponding to the respective fields of the instruction word.

RUR is an assembler macro for RUR.*, which provides compatibility with the older version of the instruction.

Operation

```
AR[r] ← user_register[st]
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor*Disabled) if Coprocessor Option

S8I Store 8-bit

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 0 0	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
S8I at, as, 0..255
```

Description

S8I is an 8-bit store from address register at to memory. It forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word. Therefore, the offset has a range from 0 to 255. Eight bits (1 byte) from the least significant quarter of address register at are written to memory at the physical address.

If the Region Translation Option (page 156) or the MMU Option (page 158)is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Operation

```
$vAddr \leftarrow$ AR[s] + ($0^{24}$||imm8)
Store8 ($vAddr$, AR[t]$_{7..0}$)
```

Exceptions

- Memory Group (see page 244)
- GenExcep(StoreProhibitedCause) if Region Protection Option or MMU Option
- DebugExcep(DBREAK) if Debug Option

Store 16-bit S16I

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 0 1	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
S16I at, as, 0..510
```

Description

`S16I` is a 16-bit store from address register `at` to memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by one. Therefore, the offset can specify multiples of two from zero to 510. Sixteen bits (two bytes) from the least significant half of the register are written to memory at the physical address.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the least significant bit of the address is ignored. A reference to an odd address produces the same result as a reference to the address, minus one. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, `S16I` calculates the sum of address register `as` and the `imm8` field of the instruction word times two. Therefore, the machine-code offset is in terms of 16-bit (2 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by two.

Operation

```
$vAddr \leftarrow \mathrm{AR[s]} + (0^{23} \| \mathrm{imm8} \| 0)$
Store16 ($vAddr$, $\mathrm{AR[t]}_{15..0}$)
```

Exceptions

- Memory Store Group (see page 245)

S32C1I Store 32-bit Compare Conditional

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 1 1 0	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Conditional Store Option (See Section 4.3.13 on page 77)

Assembler Syntax

```
S32C1I at, as, 0..1020
```

Description

S32C1I is a conditional store instruction intended for updating synchronization variables in memory shared between multiple processors. It may also be used to atomically update variables shared between different interrupt levels or other pairs of processes on a single processor. S32C1I attempts to store the contents of address register at to the virtual address formed by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. If the old contents of memory at the physical address equals the contents of the SCOMPARE1 Special Register, the new data is written; otherwise the memory is left unchanged. In either case, the value read from the location is written to address register at. The memory read, compare, and write may take place in the processor or the memory system, depending on the Xtensa ISA implementation, as long as these operations exclude other writes to this location. See Section 4.3.13 “Conditional Store Option” on page 77 for more information on where the atomic operation takes place.

From a memory ordering point of view, the atomic pair of accesses has the characteristics of both an acquire and a release. That is, the atomic pair of accesses does not begin until all previous loads, stores, acquires, and releases have performed. The atomic pair must perform before any following load, store, acquire, or release may begin.

If the Region Translation Option (page 156) or the MMU Option (page 158)is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

S32C1I does both a load and a store when the store is successful. However, memory protection tests check for store capability and the instruction may raise a StoreProhibitedCause exception, but will never raise a LoadProhibited Cause exception.

Assembler Note

To form a virtual address, S32C1I calculates the sum of address register as and the imm8 field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^{2}$)
(mem32, error) ← Store32C1 (vAddr, AR[t], SCOMPARE1)
if error then
     EXCVADDR ← vAddr
     Exception (LoadStoreError)
else
     AR[t] ← mem32
endif
```

Exceptions

- Memory Store Group (see page 245)

S32E Store 32-bit for Window Exceptions

Instruction Word (RRI4)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	1 0 0 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Windowed Register Option (See Section 4.7.1 on page 180)

Assembler Syntax

```
S32E at, as, -64..-4
```

Description

S32E is a 32-bit store instruction similar to S32I, but with semantics required by window overflow and window underflow exception handlers. In particular, memory access checking is done with PS.RING instead of CRING, and the offset used to form the virtual address is a 4-bit one-extended immediate. Therefore, the offset can specify multiples of four from -64 to -4. In configurations without the MMU Option, there is no PS.RING and S32E is similar to S32I with a negative offset.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

S32E is a privileged instruction.

Assembler Note

To form a virtual address, S32E calculates the sum of address register as and the r field of the instruction word times four (and one extended). Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Store 32-bit for Window Exceptions S32E

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedInstructionCause)
else
    vAddr ← AR[s] + ($1^{26}||r||0^{2}$)
    ring ← if MMU Option then PS.RING else 0
    Store32Ring (vAddr, AR[t], ring)
endif
```

Exceptions

- Memory Store Group (see page 245)
- GenExcep(PrivilegedCause) if Exception Option

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 1 0	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
    S32I at, as, 0..1020
```

Description

S32I is a 32-bit store from address register at to memory. It forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. The data to be stored is taken from the contents of address register at and written to memory at the physical address.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

S32I is one of only a few memory reference instructions that can access instruction RAM.

Assembler Note

The assembler may convert S32I instructions to S32I.N when the Code Density Option is enabled and the imm8 operand falls within the available range. Prefixing the S32I instruction with an underscore (_S32I) disables this optimization and forces the assembler to generate the wide form of the instruction.

Store 32-bit S32I

To form a virtual address, S32I calculates the sum of address register as and the imm8 field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + $(0^{22}||\text{imm8}||0^{2})$
Store32 (vAddr, AR[t])
```

Exceptions

- Memory Store Group (see page 245)

Instruction Word (RRRN)

15 12	11 8	7 4	3 0
imm4	s	t	1 0 0 1
4	4	4	4

Required Configuration Option

Code Density Option (See Section 4.3.1 on page 53)

Assembler Syntax

```
S32I.N at, as, 0..60
```

Description

S32I.N is similar to S32I, but has a 16-bit encoding and supports a smaller range of offset values encoded in the instruction word.

S32I.N is a 32-bit store to memory. It forms a virtual address by adding the contents of address register as and an 4-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 60. The data to be stored is taken from the contents of address register at and written to memory at the physical address.

S32I.N is one of only a few memory reference instructions that can access instruction RAM.

If the Region Translation Option (page 156) or the MMU Option (page 158)is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Options, such an access raises an exception.

Assembler Note

The assembler may convert S32I.N instructions to S32I. Prefixing the S32I.N instruction with an underscore (_S32I.N) disables this optimization and forces the assembler to generate the narrow form of the instruction.

To form a virtual address, S32I.N calculates the sum of address register as and the imm4 field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
    $vAddr \leftarrow$ AR[s] + ($0^{26}$||imm4||$0^{2}$)
    Store32 (vAddr, AR[t])
```

Exceptions

- Memory Store Group (see page 245)

S32RI Store 32-bit Release

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 1 1 1	s	t	0 0 1 0
8	4	4	4	4

Required Configuration Option

Multiprocessor Synchronization Option (See Section 4.3.12 on page 74)

Assembler Syntax

```
S32RI at, as, 0..1020
```

Description

`S32RI` is a store barrier and 32-bit store from address register `at` to memory. `S32RI` stores to synchronization variables, which signals that previously written data is "released" for consumption by readers of the synchronization variable. This store will not perform until all previous loads, stores, acquires, and releases have performed. This ensures that any loads of the synchronization variable that see the new value will also find all previously written data available as well.

`S32RI` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. `S32RI` waits for previous loads, stores, acquires, and releases to be performed, and then the data to be stored is taken from the contents of address register `at` and written to memory at the physical address. Because the method of waiting is implementation dependent, this is indicated in the operation section below by the implementation function `release`.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without theUnaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, `S32RI` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^{2}$)
release()
Store32 (vAddr, AR[t])
```

Exceptions

- Memory Store Group (see page 245)

SDCT

Store Data Cache Tag

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 1	1 0 0 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Data Cache Test Option (See Section 4.5.6 on page 121)

Assembler Syntax

```
SDCT at, as
```

Description

SDCT is not part of the Xtensa Instruction Set Architecture, but is instead specific to an implementation. That is, it may not exist in all implementations of the Xtensa ISA.

SDCT is intended for writing the RAM array that implements the data cache tags as part of manufacturing test.

SDCT uses the contents of address register as to select a line in the data cache and writes the contents of address register at to the tag associated with that line. The value written from at is described under Cache Tag Format in Section 4.5.1.2 on page 112.

SDCT is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     $index \leftarrow AR[s]_{dih..dil}$
     $DataCacheTag[index] \leftarrow AR[t]$
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)
```

The cache line specified by index $\mathtt{AR[s]}_{x-1..z}$ in a direct-mapped cache or way $\mathtt{AR[s]}_{x-1..y}$ and index $\mathtt{AR[s]}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing.

SEXT Sign Extend

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 0	0 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Miscellaneous Operations Option (See Section 4.3.8 on page 62)

Assembler Syntax

```
SEXT ar, as, 7..22
```

Description

`SEXT` takes the contents of address register `as` and replicates the bit specified by its immediate operand (in the range 7 to 22) to the high bits and writes the result to address register `ar`. The input can be thought of `as` an `imm+1` bit value with the high bits irrelevant and this instruction produces the 32-bit sign-extension of this value.

Assembler Note

The immediate values accepted by the assembler are 7 to 22. The assembler encodes these in the `t` field of the instruction using 0 to 15.

Operation

```
$b \leftarrow \texttt{t+7}$
$\texttt{AR[r]} \leftarrow \texttt{AR[s]}_b{}^{31-b} \| \texttt{AR[s]}_{b..0}$
```

Exceptions

- EveryInstR Group (see page 244)

Store Instruction Cache Tag

SICT

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 1	0 0 0 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Instruction Cache Test Option (See Section 4.5.3 on page 116)

Assembler Syntax

```
SICT at, as
```

Description

`SICT` is not part of the Xtensa Instruction Set Architecture, but is instead specific to an implementation. That is, it may not exist in all implementations of the Xtensa ISA.

`SICT` is intended for writing the RAM array that implements the instruction cache tags as part of manufacturing test.

`SICT` uses the contents of address register `as` to select a line in the instruction cache, and writes the contents of address register `at` to the tag associated with that line. The value written from `at` is described under Cache Tag Format in Section 4.5.1.2 on page 112.

`SICT` is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     index ← $AR[s]_{iih..iil}$
     InstCacheTag[index] ← AR[t]
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(InstCacheBytes))
y ← log2(InstCacheBytes ÷ InstCacheWayCount)
z ← log2(InstCacheLineBytes)
```

The cache line specified by index $\mathtt{AR[s]}_{x-1..z}$ in a direct-mapped cache or way $\mathtt{AR[s]}_{x-1..y}$ and index $\mathtt{AR[s]}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing.

Store Instruction Cache Word SICW

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 1	0 0 1 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Instruction Cache Test Option (See Section 4.5.3 on page 116)

Assembler Syntax

```
SICW at, as
```

Description

SICW is not part of the Xtensa Instruction Set Architecture, but is instead specific to an implementation. That is, it may not exist in all implementations of the Xtensa ISA.

SICW is intended for writing the RAM array that implements the instruction cache as part of manufacturing tests.

SICW uses the contents of address register as to select a line in the instruction cache, and writes the contents of address register at to the data associated with that line.

SICW is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     index ← $AR[s]_{iih..iiw}$
     InstCacheData [index] ← AR[t]
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(InstCacheBytes))
y ← log2(InstCacheBytes ÷ InstCacheWayCount)
z ← log2(InstCacheLineBytes)
```

The cache line specified by index $AR[s]_{x-1..z}$ in a direct-mapped cache or way $AR[s]_{x-1..y}$ and index $AR[s]_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing. Within the cache line, $AR[s]_{z-1..2}$ is used to determine which 32-bit quantity within the line is written.

The width of the instruction cache RAM may be more than 32 bits depending on the configuration. In that case, some implementations may write the same data replicated enough times to fill the entire width of the RAM.

Simulator Call — SIMCALL

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 1 0 1	0 0 0 1	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Xtensa Instruction Set Simulator only — illegal in hardware

Assembler Syntax

```
SIMCALL
```

Description

`SIMCALL` is not implemented by any Xtensa processor. Processors raise an illegal instruction exception for this opcode. It is implemented by the Xtensa Instruction Set Simulator only to allow simulated programs to request services of the simulator host processor. See the *Xtensa Instruction Set Simulator (ISS) User's Guide*.

The value in address register `a2` is the request code. Most codes request host system call services while others are used for special purposes such as debugging. Arguments needed by host system calls will be found in `a3`, `a4`, and `a5` and a return code will be stored to `a2` and an error number to `a3`.

Operation

See the *Xtensa Instruction Set Simulator (ISS) User's Guide*.

Exceptions

- EveryInst Group (see page 244)
- GenExcep(IllegalInstructionCause) if Exception Option

SLL Shift Left Logical

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 0	0 0 0 1	r	s	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SLL ar, as
```

Description

`SLL` shifts the contents of address register `as` left by the number of bit positions specified (as 32 minus number of bit positions) in the `SAR` (shift amount register) and writes the result to address register `ar`. Typically the `SSL` or `SSA8L` instructions are used to specify the left shift amount by loading `SAR` with `32-shift`. This transformation allows `SLL` to be implemented in the `SRC` funnel shifter (which only shifts right), using the `SLL` data as the most significant 32 bits and zero as the least significant 32 bits. Note the result of `SLL` is undefined if `SAR` > `32`.

Operation

```
$sa \leftarrow \mathrm{SAR}_{5..0}$
$\mathrm{AR[r]} \leftarrow (\mathrm{AR[s]} \| 0^{32})_{31+sa..sa}$
```

Exceptions

- EveryInstR Group (see page 244)

Shift Left Logical Immediate

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 sa_4	0 0 0 1	r	s	$sa_{3..0}$	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SLLI ar, as, 1..31
```

Description

`SLLI` shifts the contents of address register `as` left by a constant amount in the range `1..31` encoded in the instruction. The shift amount `sa` field is split, with bits 3..0 in bits 7..4 of the instruction word and bit 4 in bit 20 of the instruction word. The shift amount is encoded as `32−shift`. When the `sa` field is 0, the result of this instruction is undefined.

Assembler Note

The shift amount is specified in the assembly language as the number of bit positions to shift left. The assembler performs the `32-shift` calculation when it assembles the instruction word. When the immediate operand evaluates to zero, the assembler converts this instruction to an `OR` instruction to effect a register-to-register move. To disable this transformation, prefix the mnemonic with an underscore (`_SLLI`). If `imm` evaluates to zero when the mnemonic has the underscore prefix, the assembler will emit an error.

Operation

$$\mathtt{AR[r]} \leftarrow (\mathtt{AR[s]} \| 0^{32})_{\mathtt{31+sa..sa}}$$

Exceptions

- EveryInstR Group (see page 244)

SRA Shift Right Arithmetic

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 1 1	0 0 0 1	r	0 0 0 0	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SRA ar, at
```

Description

SRA arithmetically shifts the contents of address register at right, inserting the sign of at on the left, by the number of bit positions specified by SAR (shift amount register) and writes the result to address register ar. Typically the SSR or SSA8B instructions are used to load SAR with the shift amount from an address register. Note the result of SRA is undefined if SAR > 32.

Operation

```
$sa \leftarrow \mathrm{SAR}_{5..0}$
$\mathrm{AR[r]} \leftarrow ((\mathrm{AR[t]}_{31})^{32} \| \mathrm{AR[t]})_{31+sa..sa}$
```

Exceptions

- EveryInstR Group (see page 244)

Shift Right Arithmetic Immediate SRAI

Instruction Word (RRR)

23			20	19			16	15 12	11 8	7 4	3			0
0	0	1	sa_4	0	0	0	1	r	$sa_{3..0}$	t	0	0	0	0
4				4				4	4	4	4			

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SRAI ar, at, 0..31
```

Description

SRAI arithmetically shifts the contents of address register at right, inserting the sign of at on the left, by a constant amount encoded in the instruction word in the range 0..31. The shift amount sa field is split, with bits 3..0 in bits 11..8 of the instruction word, and bit 4 in bit 20 of the instruction word.

Operation

```
$AR[r] \leftarrow ((AR[t]_{31})^{32} \| AR[t])_{31+sa..sa}$
```

Exceptions

- EveryInstR Group (see page 244)

SRC Shift Right Combined

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 0	0 0 0 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SRC ar, as, at
```

Description

SRC performs a right shift of the concatenation of address registers as and at by the shift amount in SAR. The least significant 32 bits of the shift result are written to address register ar. A shift with a wider input than output is called a funnel shift. SRC directly performs right funnel shifts. Left funnel shifts are done by swapping the high and low operands to SRC and setting SAR to 32 minus the shift amount. The SSL and SSA8B instructions directly implement such SAR settings. Note the result of SRC is undefined if SAR > 32.

Operation

```
$sa \leftarrow \mathrm{SAR}_{5..0}$
$\mathrm{AR[r]} \leftarrow (\mathrm{AR[s]} \| \mathrm{AR[t]})_{31+sa..sa}$
```

Exceptions

- EveryInstR Group (see page 244)

Shift Right Logical SRL

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 1	0 0 0 1	r	0 0 0 0	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
    SRL ar, at
```

Description

SRL shifts the contents of address register at right, inserting zeros on the left, by the number of bits specified by SAR (shift amount register) and writes the result to address register ar. Typically the SSR or SSA8B instructions are used to load SAR with the shift amount from an address register. Note the result of SRL is undefined if SAR > 32.

Operation

```
    $sa \leftarrow \mathrm{SAR}_{5..0}$
    $\mathrm{AR}[r] \leftarrow (0^{32} \| \mathrm{AR}[t])_{31+sa..sa}$
```

Exceptions

- EveryInstR Group (see page 244)

SRLI Shift Right Logical Immediate

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 0 1	r	sa	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SRLI ar, at, 0..15
```

Description

`SRLI` shifts the contents of address register `at` right, inserting zeros on the left, by a constant amount encoded in the instruction word in the range `0..15`. There is no `SRLI` for shifts ≥ 16. `EXTUI` replaces these shifts.

Assembler Note

The assembler converts `SRLI` instructions with a shift amount ≥ 16 into `EXTUI`. Prefixing the `SRLI` instruction with an underscore (`_SRLI`) disables this replacement and forces the assembler to generate an error.

Operation

$$\mathtt{AR[r]} \leftarrow (0^{32} \| \mathtt{AR[t]})_{\mathtt{31+sa..sa}}$$

Exceptions

- EveryInstR Group (see page 244)

Set Shift Amount for BE Byte Shift SSA8B

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 0 0	0 0 1 1	s	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
    SSA8B as
```

Description

SSA8B sets the shift amount register (SAR) for a left shift by multiples of eight (for example, for big-endian (BE) byte alignment). The left shift amount is the two least significant bits of address register as multiplied by eight. Thirty-two minus this amount is written to SAR. Using 32 minus the left shift amount causes a funnel right shift and swapped high and low input operands to perform a left shift. SSA8B is similar to SSL, except the shift amount is multiplied by eight.

SSA8B is typically used to set up for an SRC instruction to shift bytes. It may be used with big-endian byte ordering to extract a 32-bit value from a non-aligned byte address.

Operation

```
    $\mathrm{SAR} \leftarrow 32 - (0 \| \mathrm{AR[s]}_{1..0} \| 0^3)$
```

Exceptions

- EveryInstR Group (see page 244)

SSA8L Set Shift Amount for LE Byte Shift

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 0 0	0 0 1 0	s	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SSA8L as
```

Description

SSA8L sets the shift amount register (SAR) for a right shift by multiples of eight (for example, for little-endian (LE) byte alignment). The right shift amount is the two least significant bits of address register as multiplied by eight, and is written to SAR. SSA8L is similar to SSR, except the shift amount is multiplied by eight.

SSA8L is typically used to set up for an SRC instruction to shift bytes. It may be used with little-endian byte ordering to extract a 32-bit value from a non-aligned byte address.

Operation

```
$SAR \leftarrow 0||AR[s]_{1..0}||0^3$
```

Exceptions

- EveryInstR Group (see page 244)

Set Shift Amount Immediate SSAI

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 0 0	0 1 0 0	$sa_{3..0}$	0 0 0 sa_4	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SSAI 0..31
```

Description

SSAI sets the shift amount register (SAR) to a constant. The shift amount sa field is split, with bits 3..0 in bits 11..8 of the instruction word, and bit 4 in bit 4 of the instruction word. Because immediate forms exist of most shifts (SLLI, SRLI, SRAI), this is primarily useful to set the shift amount for SRC.

Operation

```
SAR ← 0||sa
```

Exceptions

- EveryInst Group (see page 244)

SSI Store Single Immediate

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	0 1 0 0	s	t	0 0 1 1
8	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
SSI ft, as, 0..1020
```

Description

SSI is a 32-bit store from floating-point register ft to memory. It forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. The data to be stored is taken from the contents of floating-point register ft and written to memory at the physical address.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, SSI calculates the sum of address register as and the imm8 field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + ($0^{22}$||imm8||$0^{2}$)
Store32 (vAddr, FR[t])
```

Exceptions

- Memory Store Group (see page 245)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

SSIU Store Single Immediate with Update

Instruction Word (RRI8)

23 16	15 12	11 8	7 4	3 0
imm8	1 1 0 0	s	t	0 0 1 1
8	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
SSIU ft, as, 0..1020
```

Description

SSIU is a 32-bit store from floating-point register ft to memory with base address register update. It forms a virtual address by adding the contents of address register as and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. The data to be stored is taken from the contents of floating-point register ft and written to memory at the physical address. The virtual address is written back to address register as.

If the Region Translation Option (page 156) or the MMU Option (page 158) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, SSIU calculates the sum of address register as and the imm8 field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
$vAddr \leftarrow \texttt{AR[s]} + (0^{22}||\texttt{imm8}||0^{2})$
```

```
Store32 (vAddr, FR[t])
AR[s] ← vAddr
```

Exceptions

- Memory Store Group (see page 245)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

SSL Set Shift Amount for Left Shift

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 0 0	0 0 0 1	s	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SSL as
```

Description

SSL sets the shift amount register (SAR) for a left shift (for example, SLL). The left shift amount is the 5 least significant bits of address register as. 32 minus this amount is written to SAR. Using 32 minus the left shift amount causes a right funnel shift, and swapped high and low input operands to perform a left shift.

Operation

```
$sa \leftarrow \mathrm{AR}[s]_{4..0}$
$\mathrm{SAR} \leftarrow 32 - (0||sa)$
```

Exceptions

- EveryInstR Group (see page 244)

Set Shift Amount for Right Shift SSR

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 0 0	0 0 0 0	s	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SSR as
```

Description

SSR sets the shift amount register (SAR) for a right shift (for example, SRL, SRA, or SRC). The least significant five bits of address register as are written to SAR. The most significant bit of SAR is cleared. This instruction is similar to a WSR.SAR, but differs in that only $AR[s]_{4..0}$ is used, instead of $AR[s]_{5..0}$.

Operation

```
$sa \leftarrow AR[s]_{4..0}$
$SAR \leftarrow 0 \| sa$
```

Exceptions

- EveryInstR Group (see page 244)

SSX Store Single Indexed

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	1 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
SSX fr, as, at
```

Description

SSX is a 32-bit store from floating-point register ft to memory. It forms a virtual address by adding the contents of address register as and the contents of address register at. The data to be stored is taken from the contents of floating-point register fr and written to memory at the physical address.

If the Region Translation Option (page 156) or the MMU Option (page 158)is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

```
vAddr ← AR[s] + (AR[t])
Store32 (vAddr, FR[r])
```

Exceptions

- Memory Store Group (see page 245)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Store Single Indexed with Update SSXU

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	1 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
SSXU fr, as, at
```

Description

`SSXU` is a 32-bit store from floating-point register `ft` to memory with base address register update. It forms a virtual address by adding the contents of address register `as` and the contents of address register `at`. The data to be stored is taken from the contents of floating-point register `fr` and written to memory at the physical address. The virtual address is written back to address register `as`.

If the Region Translation Option (page 156) or the MMU Option (page 158)is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see Section 4.4.1.5 on page 89).

Without the Unaligned Exception Option (page 99), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

```
vAddr ← AR[s] + (AR[t])
Store32 (vAddr, FR[r])
AR[s] ← vAddr
```

Exceptions

- Memory Store Group (see page 245)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

SUB Subtract

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 0 0	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SUB ar, as, at
```

Description

SUB calculates the two’s complement 32-bit difference of address registers as and at. The low 32 bits of the difference are written to address register ar. Arithmetic overflow is not detected.

Operation

```
AR[r] ← AR[s] − AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

Subtract Single SUB.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 1	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
SUB.S fr, fs, ft
```

Description

`SUB.S` computes the IEEE754 single-precision difference of the contents of floating-point registers `fs` and `ft` and writes the result to floating-point register `fr`.

Operation

```
FR[r] ← FR[s] $-_{s}$ FR[t]
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

SUBX2 Subtract with Shift by 1

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 0 1	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SUBX2 ar, as, at
```

Description

SUBX2 calculates the two's complement 32-bit difference of address register as shifted left by 1 bit and address register at. The low 32 bits of the difference are written to address register ar. Arithmetic overflow is not detected.

SUBX2 is frequently used as part of sequences to multiply by small constants.

Operation

```
$AR[r] \leftarrow (AR[s]_{30..0} || 0) - AR[t]$
```

Exceptions

- EveryInstR Group (see page 244)

Subtract with Shift by 2 SUBX4

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 0	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SUBX4 ar, as, at
```

Description

SUBX4 calculates the two's complement 32-bit difference of address register as shifted left by two bits and address register at. The low 32 bits of the difference are written to address register ar. Arithmetic overflow is not detected.

SUBX4 is frequently used as part of sequences to multiply by small constants.

Operation

```
$AR[r] \leftarrow (AR[s]_{29..0} \| 0^2) - AR[t]$
```

Exceptions

- EveryInstR Group (see page 244)

SUBX8 Subtract with Shift by 3

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
SUBX8 ar, as, at
```

Description

SUBX8 calculates the two’s complement 32-bit difference of address register as shifted left by three bits and address register at. The low 32 bits of the difference are written to address register ar. Arithmetic overflow is not detected.

SUBX8 is frequently used as part of sequences to multiply by small constants.

Operation

```
$AR[r] \leftarrow (AR[s]_{28..0} \| 0^3) - AR[t]$
```

Exceptions

- EveryInstR Group (see page 244)

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 1 0 1	0 0 0 0	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Exception Option (See Section 4.4.1 on page 82)

Assembler Syntax

```
SYSCALL
```

Description

When executed, the `SYSCALL` instruction raises a system-call exception, redirecting execution to an exception vector (see Section 4.4.1 on page 82). Therefore, `SYSCALL` instructions never complete. `EPC[1]` contains the address of the `SYSCALL` and `ICOUNT` is not incremented. The system call handler should add 3 to `EPC[1]` before returning from the exception to continue execution.

The program may pass parameters to the system-call handler in the registers. There are no bits in `SYSCALL` instruction reserved for this purpose. See Section 8.2.2 “System Calls” on page 597 for a description of software conventions for system call parameters.

Operation

```
Exception (SyscallCause)
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(SyscallCause) if Exception Option

TRUNC.S Truncate Single to Fixed

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 0 0 1	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
TRUNC.S ar, fs, 0..15
```

Description

TRUNC.S converts the contents of floating-point register fs from single-precision to signed integer format, rounding toward 0. The single-precision value is first scaled by a power of two constant value encoded in the t field, with 0..15 representing 1.0, 2.0, 4.0, …, 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for t=0, and moves to the left as t increases until for t=15 there are 15 fractional bits represented in the fixed point number. For positive overflow (value ≥ 32'h7fffffff), positive infinity, or NaN, 32'h7fffffff is returned; for negative overflow (value ≤ 32'h80000000) or negative infinity, 32'h80000000 is returned. The result is written to address register ar.

Operation

```
AR[r] ← $\mathrm{trunc}_S(\mathrm{FR}[s] \times_S \mathrm{pow}_S(2.0,t))$
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Compare Single Unordered or Equal UEQ.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 1	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
UEQ.S br, fs, ft
```

Description

UEQ.S compares the contents of floating-point registers fs and ft. If the values are equal or unordered then Boolean register br is set to 1, otherwise br is set to 0. According to IEEE754, +0 and –0 compare as equal. IEEE754 floating-point values are unordered if either of them is a NaN.

Operation

```
$BR_r \leftarrow$ isNaN(FR[s]) or isNaN(FR[t]) or (FR[s] $=_s$ FR[t])
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

UFLOAT.S Convert Unsigned Fixed to Single

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 0 1	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
UFLOAT.S fr, as, 0..15
```

Description

`UFLOAT.S` converts the contents of address register `as` from unsigned integer to single-precision format, rounding according to the current rounding mode. The converted integer value is then scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 0.5, 0.25, …, $1.0 \div_s 32768.0$. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0`, and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. The result is written to floating-point register `fr`.

Operation

```
FR[r] ← $\text{ufloat}_s$(AR[s]) $\times_s$ $\text{pow}_s$(2.0,-t))
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Compare Single Unord or Less Than or Equal ULE.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 1	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
ULE.S br, fs, ft
```

Description

ULE.S compares the contents of floating-point registers fs and ft. If the contents of fs are less than or equal to or unordered with the contents of ft, then Boolean register br is set to 1, otherwise br is set to 0. IEEE754 specifies that +0 and –0 compare as equal. IEEE754 floating-point values are unordered if either of them is a NaN.

Operation

```
$BR_r \leftarrow$ isNaN(FR[s]) or isNaN(FR[t]) or (FR[s] $\le_s$ FR[t])
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

ULT.S Compare Single Unordered or Less Than

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
ULT.S br, fs, ft
```

Description

`ULT.S` compares the contents of floating-point registers `fs` and `ft`. If the contents of `fs` are less than or unordered with the contents of `ft`, then Boolean register `br` is set to 1, otherwise `br` is set to 0. IEEE754 specifies that +0 and −0 compare as equal. IEEE754 floating-point values are unordered if either of them is a NaN.

Operation

```
$BR_r$ ← isNaN(FR[s]) or isNaN(FR[t]) or (FR[s] $<_s$ FR[t])
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Unsigned Multiply UMUL.AA.*

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 1 1	0 0 half	0 0 0 0	s	t	0 1 0 0
4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See Section 4.3.7 on page 60)

Assembler Syntax

```
UMUL.AA.* as, at
```

Where * expands as follows:

```
UMUL.AA.LL - for (half=0)
UMUL.AA.HL - for (half=1)
UMUL.AA.LH - for (half=2)
UMUL.AA.HH - for (half=3)
```

Description

`UMUL.AA.*` performs an unsigned multiply of half of each of the address registers `as` and `at`, producing a 32-bit result. The result is zero-extended to 40 bits and written to the MAC16 accumulator.

Operation

```
m1 ← if $half_0$ then $AR[s]_{31..16}$ else $AR[s]_{15..0}$
m2 ← if $half_1$ then $AR[t]_{31..16}$ else $AR[t]_{15..0}$
ACC ← $(0^{24}||m1) \times (0^{24}||m2)$
```

Exceptions

- EveryInstR Group (see page 244)

UN.S Compare Single Unordered

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 1	1 0 1 1	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
UN.S br, fs, ft
```

Description

`UN.S` sets Boolean register `br` to 1 if the contents of either floating-point register `fs` or `ft` is a IEEE754 NaN; otherwise `br` is set to 0.

Operation

```
BR_r ← isNaN(FR[s]) or isNaN(FR[t])
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

Truncate Single to Fixed Unsigned UTRUNC.S

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 0	1 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
UTRUNC.S ar, fs, 0..15
```

Description

`UTRUNC.S` converts the contents of floating-point register `fs` from single-precision to unsigned integer format, rounding toward 0. The single-precision value is first scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 2.0, 4.0, …, 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0`, and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. For positive overflow (value ≥ `32'hffffffff`), positive infinity, or NaN, `32'hffffffff` is returned; for negative numbers or negative infinity, `32'h80000000` is returned. The result is written to address register `ar`.

Operation

```
$AR[r] \leftarrow utrunc_S(FR[s] \times_S pow_S(2.0,t))$
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

WAITI Wait for Interrupt

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 0	0 0 0 0	0 1 1 1	imm4	0 0 0 0	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Interrupt Option (See Section 4.4.4 on page 100)

Assembler Syntax

```
WAITI 0..15
```

Description

`WAITI` sets the interrupt level in `PS.INTLEVEL` to `imm4` and then, on some Xtensa ISA implementations, suspends processor operation until an interrupt occurs. `WAITI` is typically used in an idle loop to reduce power consumption. `CCOUNT` continues to increment during suspended operation, and a `CCOMPARE` interrupt will wake the processor.

When an interrupt is taken during suspended operation, `EPC[i]` will have the address of the instruction following `WAITI`. An implementation is not required to enter suspended operation and may leave suspended operation and continue execution at the following instruction at any time. Usually, therefore, the `WAITI` instruction should be within a loop.

The combination of setting the interrupt level and suspending operation avoids a race condition where an interrupt between the interrupt level setting and the suspension of operation would be ignored until a second interrupt occurred.

`WAITI` is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     PS.INTLEVEL ← imm4
endif
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	0 0 0 0	1 1 1 0	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Region Translation Option (page 156) or the MMU Option (page 158)

Assembler Syntax

```
WDTLB at, as
```

Description

`WDTLB` uses the contents of address register `as` to specify a data TLB entry and writes the contents of address register `at` into that entry. See Section 4.6 on page 138 for information on the address and result register formats for specific memory protection and translation options. The point at which the data TLB write is effected is implementation-specific. Any translation that would be affected by this write before the execution of a `DSYNC` instruction is therefore undefined.

`WDTLB` is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     (vpn, ei, wi) ← SplitDataTLBEntrySpec(AR[s])
     (ppn, sr, ring, ca) ← SplitDataEntry(wi, AR[t])
     DataTLB[wi][ei].ASID ← ASID(ring)
     DataTLB[wi][ei].VPN ← vpn
     DataTLB[wi][ei].PPN ← ppn
     DataTLB[wi][ei].SR ← sr
     DataTLB[wi][ei].CA ← ca
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

WER Write External Register

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 0 0	0 1 1 1	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
WER at, as
```

Description

WER writes one of a set of "External Registers". It is in some ways similar to the WSR.* instruction except that the registers being written are not defined by the Xtensa ISA and are conceptually outside the processor core. They are written through processor ports.

Address register as is used to determine which register is to be written and address register at provides the write data. When no External Register is addressed by the value in address register as, no write occurs. The entire address space is reserved for use by Tensilica. RER and WER are managed by the processor core so that the requests appear on the processor ports in program order. External logic is responsible for extending that order to the registers themselves.

WER is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     Write External Register as defined outside the processor.
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Move AR to FR WFR

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
1 1 1 1	1 0 1 0	r	s	0 1 0 1	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See Section 4.3.11 on page 67)

Assembler Syntax

```
WFR fr, as
```

Description

`WFR` moves the contents of address register `as` to floating-point register `fr`. The move is non-arithmetic; no floating-point exceptions are raised.

Operation

```
FR[r] ← AR[s]
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor0Disabled) if Coprocessor Option

WITLB   Write Instruction TLB Entry

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 1	0 0 0 0	0 1 1 0	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Region Translation Option (page 156) or the MMU Option (page 158)

Assembler Syntax

```
WITLB at, as
```

Description

`WITLB` uses the contents of address register `as` to specify an instruction TLB entry and writes the contents of address register `at` into that entry. See Section 4.6 on page 138 for information on the address and result register formats for specific memory protection and translation options. The point at which the instruction TLB write is effected is implementation-specific. Any translation that would be affected by this write before the execution of an `ISYNC` instruction is therefore undefined.

`WITLB` is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     (vpn, ei, wi) ← SplitInstTLBEntrySpec(AR[s])
     (ppn, sr, ring, ca) ← SplitInstEntry(wi, AR[t])
     InstTLB[wi][ei].ASID ← ASID(ring)
     InstTLB[wi][ei].VPN ← vpn
     InstTLB[wi][ei].PPN ← ppn
     InstTLB[wi][ei].SR ← sr
     InstTLB[wi][ei].CA ← ca
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(PrivilegedCause) if Exception Option

Write Special Register WSR.*

Instruction Word (RSR)

23 20	19 16	15 8	7 4	3 0
0 0 0 1	0 0 1 1	sr	t	0 0 0 0
4	4	8	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
WSR.* at

WSR at, *

WSR at, 0..255
```

Description

WSR.* writes the special registers that are described in Section 3.8.10 “Processor Control Instructions” on page 45. See Section 5.3 on page 208 for more detailed information on the operation of this instruction for each Special Register.

The contents of address register at are written to the special register designated by the 8-bit sr field of the instruction word. The name of the Special Register is used in place of the ‘*’ in the assembler syntax above and the translation is made to the 8-bit sr field by the assembler.

WSR is an assembler macro for WSR.* that provides compatibility with the older versions of the instruction containing either the name or the number of the Special Register.

The point at which WSR.* to certain registers affects subsequent instructions is not always defined (SAR and ACC are exceptions). In these cases, the Special Register Tables in Section 5.3 on page 208 explain how to ensure the effects are seen by a particular point in the instruction stream (typically involving the use of one of the ISYNC, RSYNC, ESYNC, or DSYNC instructions). A WSR.* followed by an RSR.* to the same register should be separated with an ESYNC to guarantee the value written is read back. A WSR.PS followed by RSIL also requires an ESYNC.

WSR.* with Special Register numbers ≥ 64 is privileged. A WSR.* for an unconfigured register generally will raise an illegal instruction exception.

Operation

```
sr ← if msbFirst then s||r else r||s
if sr ≥ 64 and CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     see the Special Register Tables in Section 5.3 on page 208
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(IllegalInstructionCause) if Exception Option
- GenExcep(PrivilegedCause) if Exception Option

Write User Register

WUR.*

Instruction Word (RSR)

23	20	19	16	15	8	7	4	3	0
1 1 1 1		0 0 1 1		sr		t		0 0 0 0	
4		4		8		4		4	

Required Configuration Option

No Option - instructions created from the TIE language (See Section 4.3.9.2 “Coprocessor Context Switch” on page 64)

Assembler Syntax

```
WUR.* at

WUR at,*
```

Description

`WUR.*` writes TIE state that has been grouped into 32-bit quantities by the TIE `user_register` statement. The name in the `user_register` statement replaces the “*” in the instruction name and causes the correct register number to be placed in the `st` field of the encoded instruction. The contents of address register `at` are written to the TIE `user_register` designated by the 8-bit `sr` field of the instruction word.

`WUR` is an assembler macro for `WUR.*` that provides compatibility with the older version of the instruction.

Operation

```
user_register[sr] ← AR[t]
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(Coprocessor*Disabled) if Coprocessor Option

XOR Bitwise Logical Exclusive Or

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 0 1 1	0 0 0 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50)

Assembler Syntax

```
XOR ar, as, at
```

Description

`XOR` calculates the bitwise logical exclusive or of address registers `as` and `at`. The result is written to address register `ar`.

Operation

```
AR[r] ← AR[s] xor AR[t]
```

Exceptions

- EveryInstR Group (see page 244)

Boolean Exclusive Or XORB

Instruction Word (RRR)

23 20	19 16	15 12	11 8	7 4	3 0
0 1 0 0	0 0 1 0	r	s	t	0 0 0 0
4	4	4	4	4	4

Required Configuration Option

Boolean Option (See Section 4.3.10 on page 65)

Assembler Syntax

```
    XORB br, bs, bt
```

Description

`XORB` performs the logical exclusive or of Boolean registers `bs` and `bt` and writes the result to Boolean register `br`.

When the sense of one of the source Booleans is inverted (0 → true, 1 → false), use an inverted test of the result. When the sense of both of the source Booleans is inverted, use a non-inverted test of the result.

Operation

```
    $\mathrm{BR}_r \leftarrow \mathrm{BR}_s$ xor $\mathrm{BR}_t$
```

Exceptions

- EveryInstR Group (see page 244)

Instruction Word (RSR)

23 20	19 16	15 8	7 4	3 0
0 1 1 0	0 0 0 1	sr	t	0 0 0 0
4	4	8	4	4

Required Configuration Option

Core Architecture (See Section 4.2 on page 50) (added in T1040)

Assembler Syntax

```
XSR.* at

XSR at, *

XSR at, 0..255
```

Description

XSR.* simultaneously reads and writes the special registers that are described in Section 3.8.10 “Processor Control Instructions” on page 45. See Section 5.3 on page 208 for more detailed information on the operation of this instruction for each Special Register.

The contents of address register at and the Special Register designated by the immediate in the 8-bit sr field of the instruction word are both read. The read address register value is then written to the Special Register, and the read Special Register value is written to at. The name of the Special Register is used in place of the ‘*’ in the assembler syntax above and the translation is made to the 8-bit sr field by the assembler.

XSR is an assembler macro for XSR.*, which provides compatibility with the older versions of the instruction containing either the name or the number of the Special Register.

The point at which XSR.* to certain registers affects subsequent instructions is not always defined (SAR and ACC are exceptions). In these cases, the Special Register Tables in Section 5.3 on page 208 explain how to ensure the effects are seen by a particular point in the instruction stream (typically involving the use of one of the ISYNC, RSYNC, ESYNC, or DSYNC instructions). An XSR.* followed by an RSR.* to the same register should be separated with an ESYNC to guarantee the value written is read back. An XSR.PS followed by RSIL also requires an ESYNC. In general, the restrictions on XSR.* include the union of the restrictions of the corresponding RSR.* and WSR.*.

XSR.* with Special Register numbers ≥ 64 is privileged. An XSR.* for an unconfigured register generally will raise an illegal instruction exception.

Operation

```
sr ← if msbFirst then s||r else r||s
if sr ≥ 64 and CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     t0 ← AR[t]
     t1 ← see RSR frame of the Tables in Section 5.3 on page 208
     see WSR frame of the Tables in Section 5.3 on page 208 ← t0
     AR[t] ← t1
endif
```

Exceptions

- EveryInstR Group (see page 244)
- GenExcep(IllegalInstructionCause) if Exception Option
- GenExcep(PrivilegedCause) if Exception Option

7. Instruction Formats and Opcodes

7.1 Formats

The following sections show the named opcode formats for instruction encodings. The field names in these formats are used in the opcode tables in Section 7.3.1. The format names are used throughout this document. Each chart shows both big-endian and little-endian encodings with bits numbered appropriately for that endianness. The vertical bars in the formats indicate the points at which the opcode is separated, reversed in order, and reassembled to arrive at the opposite endianness format.

7.1.1 RRR

	0 3	4 7	8 11	12 15	16 19	20 23
Big End.	op0	t	s	r	op1	op2
	4	4	4	4	4	4
	23 20	19 16	15 12	11 8	7 4	3 0
Little End.	op2	op1	r	s	t	op0
	4	4	4	4	4	4

7.1.2 RRI4

	0 3	4 7	8 11	12 15	16 19	20 23
Big End.	op0	t	s	r	op1	imm4
	4	4	4	4	4	4
	23 20	19 16	15 12	11 8	7 4	3 0
Little End.	imm4	op1	r	s	t	op0
	4	4	4	4	4	4

7.1.3 RRI8

	0 3	4 7	8 11	12 15	16 23
Big End.	op0	t	s	r	imm8
	4	4	4	4	8

	23 16	15 12	11 8	7 4	3 0
Little End.	imm8	r	s	t	op0
	8	4	4	4	4

7.1.4 RI16

	0 3	4 7	8 23
Big End.	op0	t	imm16
	4	4	16

	23 8	7 4	3 0
Little End.	imm16	t	op0
	16	4	4

7.1.5 RSR

	0 3	4 7	8 15	16 19	20 23
Big End.	op0	t	rs	op1	op2
	4	4	8	4	4

	23 20	19 16	15	7 4	3 0
Little End.	op2	op1	rs	t	op0
	4	4	8	4	4

7.1.6 CALL

	0 3	4 5	6 23
Big End.	op0	n	offset
	4	2	18

	23 6	5 4	3 0
Little End.	offset	n	op0
	18	2	4

7.1.7 CALLX

	0 3	4 5	6 7	8 11	12 15	16 19	20 23
Big End.	op0	n	m	s	r	op1	op2
	4	2	2	4	4	4	4

	23 20	19 16	15 12	11 8	7	4	3 0
Little End.	op2	op1	r	s	m	n	op0
	4	4	4	4	2	2	4

7.1.8 BRI8

	0 3	4 5	6 7	8 11	12 15	16 23
Big End.	op0	n	m	s	r	imm8
	4	2	2	4	4	8

	23 16	15 12	11 8	7	4	3 0
Little End.	imm8	r	s	m	n	op0
	8	4	4	2	2	4

7.1.9 BRI12

	0 3	4 5	6 7	8 11	12 23
Big End.	op0	n	m	s	imm12
	4	2	2	4	12

	23 12	11 8	7	4 3	0
Little End.	imm12	s	m	n	op0
	12	4	2	2	4

7.1.10 RRRN

	0 3	4 7	8 11	12 15
Big End.	op0	t	s	r
	4	4	4	4

	15 12	11 8	7 4	3 0
Little End.	r	s	t	op0
	4	4	4	4

7.1.11 RI7

	0 3	4 7	8 11	12 15
Big End.	op0	i imm7$_{6..4}$	s	imm7$_{3..0}$
	4	4	4	4

	15 12	11 8	7 6 4	3 0
Little End.	imm7$_{3..0}$	s	i imm7$_{6..4}$	op0
	4	4	4	4

7.1.12 RI6

	0 3	4 7	8 11	12 15
Big End.	op0	i z $imm6_{5..4}$	s	$imm6_{3..0}$
	4	4	4	4
	15 12	11 8	7 6 5 4	3 0
Little End.	$imm6_{3..0}$	s	i z $imm6_{5..4}$	op0
	4	4	4	4

7.2 Instruction Fields

Table 7–191. Uses Of Instruction Fields

Field	Definition
op0	Major opcode
op1	4-bit sub-opcode for 24-bit instructions
op2	4-bit sub-opcode for 24-bit instructions
r	`AR` target (result), `BR` target (result), 4-bit immediate, 4-bit sub-opcode
s	`AR` source, `BR` source `AR` target
t	`AR` target, `BR` target, `AR` source, `BR` source, 4-bit sub-opcode
n	Register window increment, 2-bit sub-opcode, `n`||`00` is used as a `AR` target on `CALLn`/`CALLXn`
m	2-bit sub-opcode
i	1-bit sub-opcode
z	1-bit sub-opcode
imm6	6-bit immediate (PC-relative offset)
imm7	7-bit immediate (for `MOVI.N`)
imm8	8-bit immediate

Table 7–191. Uses Of Instruction Fields (continued)

Field	Definition
imm12	12-bit immediate
imm16	16-bit immediate
offset	18-bit PC-relative offset

7.3 Opcode Encodings

The following tables show the instruction-field bit values assigned to specific opcodes. The following special notation is used:

- The table titles tell the name of the parent opcode and what table the parent is in, the formats for instructions in this table, and in parentheses at the end, what fields still vary for items listed in this table. In the upper left corner of the table is the field decoded in the table. Below it and to the right are templates which the field matches for the corresponding row or column.
- Non-italic opcodes are instructions. These have page numbers where the corresponding instruction is described in more detail.
- *Italics opcodes* are not instructions, but are parents to other opcodes. These have table numbers that show further decode into instructions or other parents to other opcodes.
- Some entries have further conditions after them such as (s=0), which means that the s field must be zero. All other opcodes are illegal; therefore another table seems unnecessary.
- The bit-range of opcodes that use more than one table entry is delimited by vertical bars.

- Subscripts on opcodes indicate the architectural option(s) in which the opcode is implemented. The subscripts and their associated architectural options are:
  - C—Instruction Cache or Data Cache Options
  - D—MAC16 Option
  - F—Floating-Point Coprocessor Option
  - I—32-Bit Integer Multiply/Divide Option
  - L—Instruction or Data Cache Index Lock Option
  - M—MMU Option
  - N—Code Density (Narrow instructions) Option
  - P—Coprocessor Option
  - S—Speculation Option
  - U—Miscellaneous Operations Option
  - W—Windowed Registers Option
  - X—Exception or Interrupt Options
  - Y—Multiprocessor Synchronization Option

7.3.1 Opcode Maps

Table 7–192. Whole Opcode Space

op0	xx00	xx01	xx10	xx11
00xx	*QRST — Table 7–193*	L32R — page 382	*LSAI — Table 7–216*	*LSCI*$_P$ *— Table 7–220*
01xx	*MAC16*$_D$ *— Table 7–221*	*CALLN — Table 7–232*	*SI — Table 7–233*	*B — Table 7–238*
10xx	L32I.N$_N$ — page 380	S32I.N$_N$ — page 512	ADD.N$_N$ — page 249	ADDI.N$_N$ — page 252
11xx	*ST2*$_N$ *— Table 7–239*	*ST3*$_N$ *— Table 7–240*	reserved	reserved

Table 7–193. QRST (from Table 7–192) Formats RRR, CALLX, and RSR (t, s, r, op2 vary)

op1	xx00	xx01	xx10	xx11
00xx	*RST0 — Table 7–194*	*RST1 — Table 7–205*	*RST2 — Table 7–209*	*RST3 — Table 7–210*
01xx	EXTUI — page 344		*CUST0 — Section 7.3.2*	*CUST1 — Section 7.3.2*
10xx	*LSCX*$_P$ *— Table 7–211*	*LSC4 — Table 7–212*	*FP0*$_F$ *— Table 7–213*	*FP1*$_F$ *— Table 7–215*
11xx	reserved	reserved	reserved	reserved

Table 7–194. RST0 (from Table 7–193) Formats RRR and CALLX (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	*ST0 — Table 7–195*	AND — page 259	OR — page 466	XOR — page 564
01xx	*ST1 — Table 7–202*	*TLB — Table 7–203*	RT0 — *Table 7–204*	reserved
10xx	ADD — page 248	ADDX2 — page 254	ADDX4 — page 255	ADDX8 — page 256
11xx	SUB — page 542	SUBX2 — page 544	SUBX4 — page 545	SUBX8 — page 546

Table 7–195. ST0 (from Table 7–194 Formats RRR and CALLX (t, s vary)

r	xx00	xx01	xx10	xx11
00xx	*SNM0 — Table 7–196*	MOVSP$_{W}$ — page 427	*SYNC — Table 7–199*	*RFEI$_{X}$ — Table 7–200*
01xx	BREAK$_{X}$ — page 293	SYSCALL$_{X}$ — page 547 (s,t=0)	RSIL$_{X}$ — page 498	WAITI$_{X}$ — page 556 (t=0)
10xx	ANY4$_{P}$ — page 262	ALL4$_{P}$ — page 257	ANY8$_{P}$ — page 263	ALL8$_{P}$ — page 258
11xx	reserved	reserved	reserved	reserved

Table 7–196. SNM0 (from Table 7–195) Format CALLX (n, s vary)

m	00	01	10	11
	ILL — page 358 (s,n=0)	reserved	*JR — Table 7–197*	*CALLX — Table 7–198*

Table 7–197. JR (from Table 7–196) Format CALLX (s varies)

n	00	01	10	11
	RET — page 478 (s=0)	RETW$_{W}$ — page 480 (s=0)	JX — page 368	reserved

Table 7–198. CALLX (from Table 7–196) Format CALLX (s varies)

n	00	01	10	11
	CALLX0 — page 304	CALLX4$_{W}$ — page 305	CALLX8$_{W}$ — page 307	CALLX12$_{W}$ — page 309

Table 7–199. SYNC (from Table 7–195) Format RRR (s varies)

t	xx00	xx01	xx10	xx11
00xx	ISYNC — page 364 (s=0)	RSYNC — page 502 (s=0)	ESYNC — page 342 (s=0)	DSYNC — page 339 (s=0)
01xx	reserved	reserved	reserved	reserved
10xx	EXCW — page 343 (s=0)	reserved	reserved	reserved
11xx	MEMW — page 409 (s=0)	EXTW — page 345 (s=0)	reserved	reserved

Table 7–200. RFEI (from Table 7–195) Format RRR (s varies)

t	xx00	xx01	xx10	xx11
00xx	*RFET*$_X$ — *Table 7–201*	*RFI*$_X$ — page 488	*RFME* — page 489 (s=0)	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–201. RFET (from Table 7–200) Format RRR (no bits vary)

s	xx00	xx01	xx10	xx11
00xx	RFE$_X$ — page 487	RFUE$_X$ — page 491	RFDE$_X$ — page 485	reserved
01xx	RFWO$_W$ — page 492	RFWU$_W$ — page 493	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–202. ST1 (from Table 7–194) Format RRR (t, s vary)

r	xx00	xx01	xx10	xx11
00xx	SSR — page 539 (t=0)	SSL — page 538 (t=0)	SSA8L — page 532 (t=0)	SSA8B — page 531 (t=0)
01xx	SSAI — page 533 (t=0)	reserved	RER — page 477	WER — page 558
10xx	ROTW$_W$ — page 496 (s=0)	reserved	reserved	reserved
11xx	reserved	*reserved*	NSA$_U$ — page 461	NSAU$_U$ — page 462

Table 7–203. TLB (from Table 7–194) Format RRR (t, s vary)

r	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	RITLB0 — page 494
01xx	*IITLB* — page 355 (t=0)	*PITLB* — page 470	WITLB — page 560	RITLB1 — page 495
10xx	reserved	reserved	reserved	RDTLB0 — page 473
11xx	IDTLB — page 348 (t=0)	PDTLB — page 469	WDTLB — page 557	RDTLB1 — page 474

Table 7–204. RT0 (from Table 7–194) Format RRR (t, r vary)

s	xx00	xx01	xx10	xx11
00xx	NEG — page 457	ABS — page 246	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–205. RST1 (from Table 7–193) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	SLLI — page 525		SRAI — page 527	
01xx	SRLI — page 530	reserved	XSR — page 566	*ACCER — Table 7–206*
10xx	SRC — page 528	SRL — page 529 (s=0)	SLL — page 524 (t=0)	SRA — page 526 (s=0)
11xx	MUL16U — page 437	MUL16S — page 436	reserved	*IMP — Table 7–207*

Table 7–206. ACCER (from Table 7–205) Format RRR (t, s vary)

op2	xx00	xx01	xx10	xx11
00xx	RER — page 477			
01xx				
10xx	WER — page 558			
11xx				

Table 7–207. IMP (from Table 7–205) Format RRR (t, s vary) (Section 7.3.3)

r	xx00	xx01	xx10	xx11
00xx	LICT — page 388	SICT — page 519	LICW — page 390	SICW — page 521
01xx	reserved	reserved	reserved	reserved
10xx	LDCT — page 384	SDCT — page 516	reserved	reserved
11xx	reserved	reserved	*RFDX — Table 7–208*	reserved

Table 7–208. RFDX (from Table 7–207) Format RRR (s varies)

t	xx00	xx01	xx10	xx11
00xx	RFDO — page 486 (s=0)	RFDD — page 484 (s=0,1)	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–209. RST2 (from Table 7–193) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	ANDB$_P$ — page 260	ANDBC$_P$ — page 261	ORB$_P$ — page 467	ORBC$_P$ — page 468
01xx	XORB$_P$ — page 565	reserved	*reserved*	reserved
10xx	MULL$_I$ — page 450	*reserved*	MULUH$_I$ — page 456	MULSH$_I$ — page 455
11xx	QUOU$_I$ — page 472	QUOS$_I$ — page 471	REMU$_I$ — page 476	REMS$_I$ — page 475

Table 7–210. RST3 (from Table 7–193) Formats RRR and RSR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	RSR — page 500	WSR — page 561	SEXT$_U$ — page 518	CLAMPS$_U$ — page 312
01xx	MIN$_U$ — page 410	MAX$_U$ — page 407	MINU$_U$ — page 411	MAXU$_U$ — page 408
10xx	MOVEQZ — page 415	MOVNEZ — page 425	MOVLTZ — page 423	MOVGEZ — page 419
11xx	MOVF$_P$ — page 417	MOVT$_P$ — page 428	RUR — page 503	WUR — page 563

Table 7–211. LSCX (from Table 7–193) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	LSX$_F$ — page 402	LSXU$_F$ — page 404	reserved	reserved
01xx	SSX$_F$ — page 540	SSXU$_F$ — page 534	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–212. LSC4 (from Table 7–193) Format RRI4 (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	L32E — page 376	reserved	reserved	reserved
01xx	S32E — page 508	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–213. FP0 (from Table 7–193) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	ADD.S$_F$ — page 250	SUB.S$_F$ — page 543	MUL.S$_F$ — page 435	*reserved*
01xx	MADD.S$_F$ — page 406	MSUB.S$_F$ — page 430	reserved	reserved
10xx	ROUND.S$_F$ — page 497	TRUNC.S$_F$ — page 548	FLOOR.S$_F$ — page 347	CEIL.S$_F$ — page 311
11xx	FLOAT.S$_F$ — page 346	UFLOAT.S$_F$ — page 550	UTRUNC.S$_F$ — page 555	*FP1OP$_F$ — Table 7–214*

Table 7–214. FP1OP (from Table 7–213) Format RRR (s, r vary)

t	xx00	xx01	xx10	xx11
00xx	MOV.S$_F$ — page 414	ABS.S$_F$ — page 247	reserved	reserved
01xx	RFR$_F$ — page 490	WFR$_F$ — page 559	NEG.S$_F$ — page 458	reserved
10xx	*reserved*	*reserved*	*reserved*	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–215. FP1 (from Table 7–193) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	reserved	UN.S$_F$ — page 554	OEQ.S$_F$ — page 463	UEQ.S$_F$ — page 549
01xx	OLT.S$_F$ — page 465	ULT.S$_F$ — page 552	OLE.S$_F$ — page 464	ULE.S$_F$ — page 551
10xx	MOVEQZ.S$_F$ — page 416	MOVNEZ.S$_F$ — page 426	MOVLTZ.S$_F$ — page 424	MOVGEZ.S$_F$ — page 420
11xx	MOVF.S$_F$ — page 418	MOVT.S$_F$ — page 429	reserved	reserved

Table 7–216. LSAI (from Table 7–192) Formats RRI8 and RRI4 (t, s, imm8 vary)

r	xx00	xx01	xx10	xx11
00xx	L8UI — page 369	L16UI — page 372	L32I — page 378	reserved
01xx	S8I — page 504	S16I — page 505	S32I — page 510	*CACHE*$_C$ — *Table 7–217*
10xx	reserved	L16SI — page 370	MOVI — page 421	L32AI$_Y$ — page 374
11xx	ADDI — page 251	ADDMI — page 253	S32C1I$_Y$ — page 506	S32RI$_Y$ — page 514

Table 7–217. CACHE (from Table 7–216) Formats RRI8 and RRI4 (s, imm8 vary)

t	xx00	xx01	xx10	xx11
00xx	DPFR$_C$ — page 331	DPFW$_C$ — page 335	DPFRO$_C$ — page 333	DPFWO$_C$ — page 337
01xx	DHWB$_C$ — page 317	DHWBI$_C$ — page 319	DHI$_C$ — page 313	DII$_C$ — page 321
10xx	*DCE*$_C$ — *Table 7–218*	reserved	reserved	reserved
11xx	IPF$_C$ — page 360	*ICE*$_C$ — *Table 7–219*	IHI$_C$ — page 349	III$_C$ — page 353

Table 7–218. DCE (from Table 7–217) Format RRI4 (s, imm4 vary)

op1	xx00	xx01	xx10	xx11
00xx	DPFL$_L$ — page 329	reserved	DHU$_L$ — page 315	DIU$_L$ — page 323
01xx	DIWB$_C$ — page 325	DIWBI$_C$ — page 327	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–219. ICE (from Table 7–217) Format RRI4 (s, imm4 vary)

op1	xx00	xx01	xx10	xx11
00xx	IPFL$_L$ — page 362	reserved	IHU$_L$ — page 351	IIU$_L$ — page 356
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–220. LSCI (from Table 7–192) Format RRI8 (t, s, imm8 vary)

r	xx00	xx01	xx10	xx11
00xx	LSI_F — page 398	reserved	reserved	reserved
01xx	SSI_F — page 534	reserved	reserved	reserved
10xx	$LSIU_F$ — page 400	reserved	reserved	reserved
11xx	$SSIU_F$ — page 536	reserved	reserved	reserved

Table 7–221. MAC16 (from Table 7–192) Format RRR (t, s, r, op1 vary)

op2	xx00	xx01	xx10	xx11
00xx	MACID — *Table 7–222*	MACCD — *Table 7–226*	MACDD — *Table 7–224*	MACAD — *Table 7–225*
01xx	MACIA — *Table 7–223*	MACCA — *Table 7–227*	MACDA — *Table 7–228*	MACAA — *Table 7–229*
10xx	MACI — *Table 7–230*	MACC — *Table 7–231*	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–222. MACID (from Table 7–221) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	MULA.DD.LL.LDINC — page 448	MULA.DD.HL.LDINC — page 448	MULA.DD.LH.LDINC — page 448	MULA.DD.HH.LDINC — page 448
11xx	reserved	reserved	reserved	reserved

Table 7–223. MACIA (from Table 7–221) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	MULA.DA.LL.LDINC — page 443	MULA.DA.HL.LDINC — page 443	MULA.DA.LH.LDINC — page 443	MULA.DA.HH.LDINC — page 443
11xx	reserved	reserved	reserved	reserved

Table 7–224. MACDD (from Table 7–221) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	MUL.DD.LL — page 434	MUL.DD.HL — page 434	MUL.DD.LH — page 434	MUL.DD.HH — page 434
10xx	MULA.DD.LL — page 445	MULA.DD.HL — page 445	MULA.DD.LH — page 445	MULA.DD.HH — page 445
11xx	MULS.DD.LL — page 454	MULS.DD.HL — page 454	MULS.DD.LH — page 454	MULS.DD.HH — page 454

Table 7–225. MACAD (from Table 7–221) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	MUL.AD.LL — page 432	MUL.AD.HL — page 432	MUL.AD.LH — page 432	MUL.AD.HH — page 432
10xx	MULA.AD.LL — page 439	MULA.AD.HL — page 439	MULA.AD.LH — page 439	MULA.AD.HH — page 439
11xx	MULS.AD.LL — page 452	MULS.AD.HL — page 452	MULS.AD.LH — page 452	MULS.AD.HH — page 452

Table 7–226. MACCD (from Table 7–221) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	MULA.DD.LL.LDDEC — page 446	MULA.DD.HL.LDDEC — page 446	MULA.DD.LH.LDDEC — page 446	MULA.DD.HH.LDDEC — page 446
11xx	reserved	reserved	reserved	reserved

Table 7–227. MACCA (from Table 7–221) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	MULA.DA.LL.LDDEC — page 441	MULA.DA.HL.LDDEC — page 441	MULA.DA.LH.LDDEC — page 441	MULA.DA.HH.LDDEC — page 441
11xx	reserved	reserved	reserved	reserved

Table 7–228. MACDA (from Table 7–221) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	MUL.DA.LL — page 433	MUL.DA.HL — page 433	MUL.DA.LH — page 433	MUL.DA.HH — page 433
10xx	MULA.DA.LL — page 440	MULA.DA.HL — page 440	MULA.DA.LH — page 440	MULA.DA.HH — page 440
11xx	MULS.DA.LL — page 453	MULS.DA.HL — page 453	MULS.DA.LH — page 453	MULS.DA.HH — page 453

Table 7–229. MACAA (from Table 7–221) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	UMUL.AA.LL — page 553	UMUL.AA.HL — page 553	UMUL.AA.LH — page 553	UMUL.AA.HH — page 553
01xx	MUL.AA.LL — page 431	MUL.AA.HL — page 431	MUL.AA.LH — page 431	MUL.AA.HH — page 431
10xx	MULA.AA.LL — page 438	MULA.AA.HL — page 438	MULA.AA.LH — page 438	MULA.AA.HH — page 438
11xx	MULS.AA.LL — page 451	MULS.AA.HL — page 451	MULS.AA.LH — page 451	MULS.AA.HH — page 451

Table 7–230. MACI (from Table 7–221) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	LDINC — page 387 (t=0)	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–231. MACC (from Table 7–221) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	LDDEC — page 386 (t=0)	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–232. CALLN (from Table 7–192) Format CALL (offset varies)

n	00	01	10	11
	CALL0 — page 297	CALL4 — page 298	CALL8 — page 300	CALL12 — page 302

Table 7–233. SI (from Table 7–192) Formats CALL, BRI8 and BRI12(offset varies)

n	00	01	10	11
	J — page 366	*BZ — Table 7–234*	*BI0 — Table 7–235*	*BI1 — Table 7–236*

Table 7–234. BZ (from Table 7–233) Format BRI12 (s, imm12 vary)

m	00	01	10	11
	BEQZ — page 274	BNEZ — page 290	BLTZ — page 286	BGEZ — page 281

Table 7–235. BI0 (from Table 7–233) Format BRI8 (s, r, imm8 vary)

m	00	01	10	11
	BEQI — page 273	BNEI — page 289	BLTI — page 283	BGEI — page 278

Table 7–236. BI1 (from Table 7–233) Formats BRI8 and BRI12 (s, r, imm8 vary)

m	00	01	10	11
	ENTRY$_W$ — page 340	*B1 — Table 7–237*	BLTUI — page 285	BGEUI — page 280

Table 7–237. B1 (from Table 7–236) Format BRI8 (s, imm8 vary)

r	xx00	xx01	xx10	xx11
00xx	BF$_P$ — page 276	BT$_P$ — page 296	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	LOOP — page 392	LOOPNEZ — page 396	LOOPGTZ — page 394	reserved
11xx	reserved	reserved	reserved	reserved

Table 7–238. B (from Table 7–192) Format RRI8 (t, s, imm8 vary)

r	xx00	xx01	xx10	xx11
00xx	BNONE — page 292	BEQ — page 272	BLT — page 282	BLTU — page 284
01xx	BALL — page 264	BBC — page 266	BBCI — page 267	
10xx	BANY — page 265	BNE — page 288	BGE — page 277	BGEU — page 279
11xx	BNALL — page 287	BBS — page 269	BBSI — page 270	

Table 7–239. ST2 (from Table 7–192) Formats RI7 and RI6 (s, r vary)

t	xx00	xx01	xx10	xx11
00xx	MOVI.N$_N$ — page 422			
01xx				
10xx	BEQZ.N$_N$ — page 275			
11xx	BNEZ.N$_N$ — page 291			

Table 7–240. ST3 (from Table 7–192) Format RRRN (t, s vary)

r	xx00	xx01	xx10	xx11
00xx	MOV.N$_N$ — page 413	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	*S3 — Table 7–241 (s=0)*

Table 7–241. S3 (from Table 7–240) Format RRRN (no fields vary)

t	xx00	xx01	xx10	xx11
00xx	RET.N$_N$ — page 479	RETW.N$_{WN}$ — page 482	BREAK.N$_N$ — page 295	NOP.N$_N$ — page 460
01xx	reserved	reserved	ILL.N$_N$ — page 359	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

7.3.2 CUST0 and CUST1 Opcode Encodings

CUST0 and CUST1 opcode encodings shown in Table 7–193 are permanently reserved for designer-defined opcodes. In the future, customers who use these spaces exclusively for their own designer-defined opcodes will be able to add new Tensilica-defined options without changing their opcodes or binary executables.

7.3.3 Cache-Option Opcode Encodings (Implementation-Specific)

The encodings for the `r` field sub-opcodes of the `IMP` family of opcodes, which are implementation-specific Cache-Option opcodes, are shown in Table 7–207. The `IMP` family of opcodes is reserved for these implementation-specific instructions. For a description of these instructions, see Chapter 6.

8. Using the Xtensa Architecture

This chapter describes Tensilica's software tool support of the Xtensa ISA and the conventions used by software.

8.1 The Windowed Register and CALL0 ABIs

The Xtensa ISA supports two different application binary interfaces (ABIs). The windowed register ABI works with the Windowed Register Option and is the default ABI. The CALL0 ABI can be used with any Xtensa processor. It does not make use of register windows, so it typically has slightly worse performance and code size than the windowed register ABI.

These two ABIs share much in common and diverge mostly in the areas of stack frame layout and general-purpose register usage. The basic data type sizes and alignments are identical, and the argument passing and return value conventions are nearly the same.

8.1.1 Windowed Register Usage and Stack Layout

Table 8–242 shows the general-purpose register usage for the windowed register ABI. Registers a0 and a1 are reserved for the return address and stack pointer, respectively. They must always contain those values, because they are used for stack unwinding in debuggers and exception handling. Incoming arguments are stored in registers a2 through a7. The location of outgoing arguments depends on the window size.

Table 8–242. Windowed Register Usage

Register	Use
a0	Return address
a1 (sp)	Stack pointer
a2 – a7	Incoming arguments
a7	Callee's stack-frame pointer (optional)

The stack frame layout for the windowed register ABI is shown in Figure 8–53. The stack grows down, from high to low addresses. The stack pointer (SP) must be aligned to 16-byte boundaries. A stack-frame pointer (FP) may (but is not required to) be allocated in register a7. For example, it may be needed when the routine contains a call to `alloca`. If a frame pointer is used, its value is equal to the original stack pointer (immediately after entry to the function), before any `alloca` space allocation.

The register-spill overflow area is equal to *N*–4 words, where *N* can be 4, 8, or 12 as determined by the largest CALL*N* or CALLX*N* in the function. For details, see "Windowed Procedure-Call Protocol" on page 187.

The stack pointer SP should only be modified by ENTRY and MOVSP instructions. If some other instruction modifies SP, any values in the register-spill area will not be moved. An exception to this rule is when setting the initial stack pointer for a new stack, where the register-spill area is guaranteed to be empty and where MOVSP cannot safely be used.

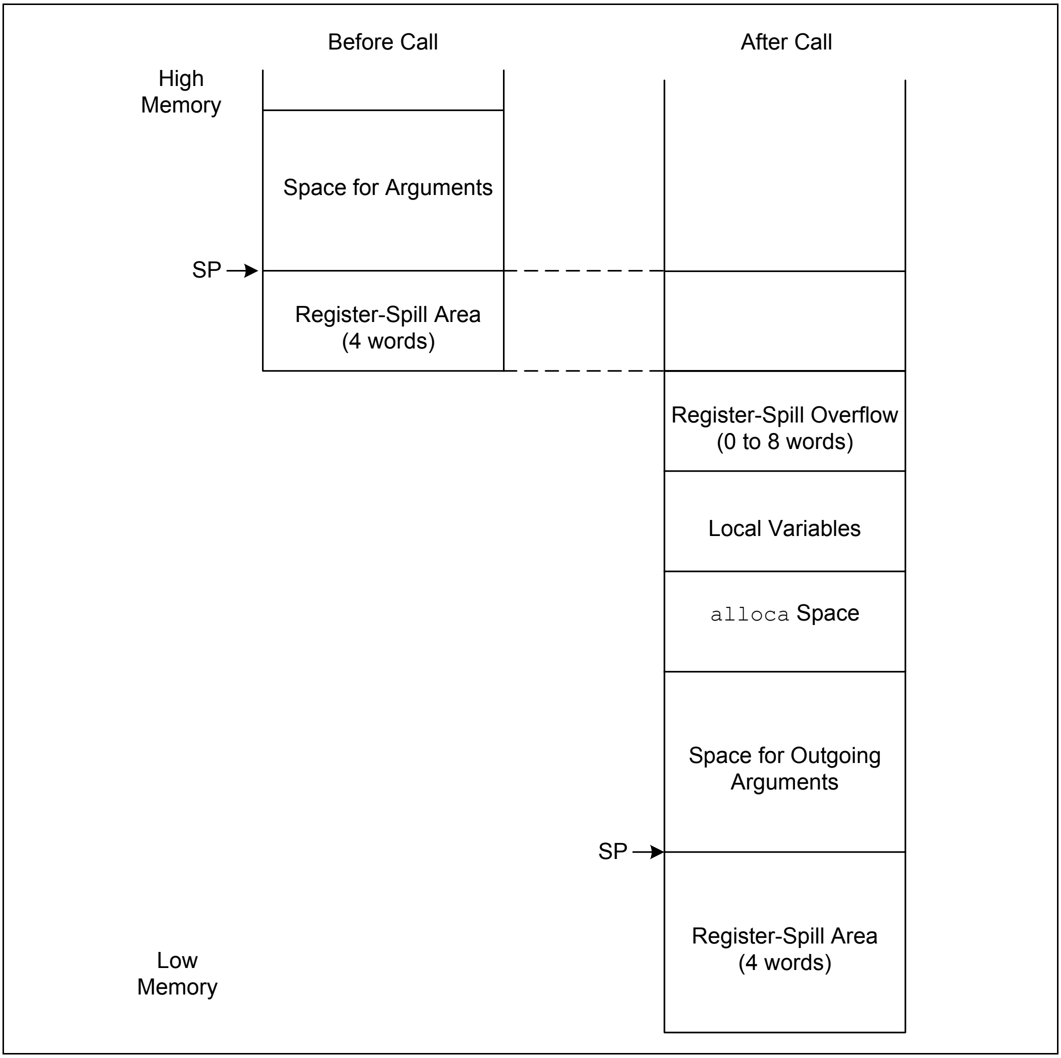

Before Call
After Call
High Memory
Space for Arguments
SP
Register-Spill Area (4 words)
Register-Spill Overflow (0 to 8 words)
Local Variables
alloca Space
Space for Outgoing Arguments
SP
Register-Spill Area (4 words)
Low Memory

Figure 8–53. Stack Frame for the Windowed Register ABI

8.1.2 CALL0 Register Usage and Stack Layout

Table 8–243 shows the general-purpose register usage for the CALL0 ABI. The stack pointer in register a1 and registers a12–a15 are callee-saved, but the rest of the registers are caller-saved. Register a0 holds the return address upon entry to a function, but unlike the windowed register ABI, it is not reserved for this purpose and may hold other values after the return address has been saved. Function arguments are passed in registers a2 through a7.

Table 8–243. CALL0 Register Usage

Register	Use
a0	Return Address
a1 (sp)	Stack Pointer (callee-saved)
a2 – a7	Function Arguments
a8	Static Chain (see Section 8.1.8)
a12 – a15	Callee-saved
a15	Stack-Frame Pointer (optional)

The stack frame layout for the CALL0 ABI is the same as for the windowed register ABI, except without the reserved register-spill areas. (Registers will need to be saved to the stack, but there is no convention for where in the frame to place that storage.) Like the windowed register ABI, the stack grows down and the stack pointer must be aligned to 16-byte boundaries. The optional stack-frame pointer is also used in the same way, but it is placed in register a15 with the CALL0 ABI.

8.1.3 Data Types and Alignment

Table 8–244 shows the data-type sizes and their alignment. The maximum alignment for user-defined types is 16 bytes.

Table 8–244. Data Types and Alignment

Data Type	Size and Alignment
`char`[1]	1 byte
`short`	2 bytes
`int`	4 bytes
`long`	4 bytes
`long long`	8 bytes
`float`	4 bytes

[1] The `char` type is unsigned by default for Xtensa processors.

[2] The `xtbool` types are only available if the Boolean registers are included in the processor configuration. See “Boolean Option” on page 65 for information about the Boolean registers.

Table 8–244. Data Types and Alignment (continued)

Data Type	Size and Alignment
`double`	8 bytes
`long double`	8 bytes
`pointer`	4 bytes
`xtbool`[2]	1 byte
`xtbool2`[2]	1 byte
`xtbool4`[2]	1 byte
`xtbool8`[2]	1 byte
`xtbool16`[2]	2 bytes
`user-defined types`	user-defined

[1] The `char` type is unsigned by default for Xtensa processors.

[2] The `xtbool` types are only available if the Boolean registers are included in the processor configuration. See "Boolean Option" on page 65 for information about the Boolean registers.

8.1.4 Argument Passing

Arguments are passed in both registers and memory. In general, the first six words of arguments go in the AR register file, and any remaining arguments go on the stack. For a `CALL`*`N`* instruction (where *N* is 0 for the CALL0 ABI, or where *N* is 4, 8, or 12 for the windowed register ABI) the caller places the first arguments in registers AR[*N*+2] through AR[*N*+7]. (Note that this implies that `CALL12` can only be used when there are two words of arguments or less; only AR[*N*+2] and AR[*N*+3] can be used when *N*=12.) The callee receives these arguments in AR[2] through AR[7].

If there are more than six words of arguments, the additional arguments are stored on the stack beginning at the caller's stack pointer and at increasingly positive offsets from the stack pointer. That is, the caller stores the seventh argument word (after the first six words in registers) at [sp + 0], the eighth word at [sp + 4], and so on. The callee can access these arguments in memory beginning at [sp + *FRAMESIZE*], where *FRAMESIZE* is the size of the callee's stack frame.

All arguments consist of an integral number of 4-byte words. Thus, the minimum argument size is one word. Integer values smaller than a word (that is, `char` and `short`) are stored in the least significant portion of the argument word, with the upper bits set to zero for unsigned values or sign-extended for signed values.

When a value larger than 4 bytes is passed in registers, the ordering of the words is the same as the byte ordering. With little endian ordering, the least significant word goes in the first register. With big endian ordering, the most significant word comes first.

Each argument must be passed entirely in registers or entirely on the stack; an argument cannot be split with some words in registers and the remainder on the stack. If an argument does not fit entirely in the remaining unused registers, it is passed on the stack and those registers remain unused.

Arguments must be properly aligned. If the type of the argument requires 4-byte or less alignment, this requirement has no effect; all arguments have at least 4-byte alignment anyway. If an argument requires 8-byte alignment and is passed in registers, the first word must be in an even-numbered register. This sometimes requires leaving an odd-numbered register unused. Similarly, if an argument requires 16-byte alignment and is passed in registers, the first word must be in the first argument register (AR[*N*+2]); otherwise, it is passed on the stack. If an argument is passed in memory, the memory location must have the alignment required by the argument type.

Structures and other aggregate types are passed by value. The preceding rules apply to structures in the same way as scalars. If a structure is small enough to be passed in registers, the words of the structure are placed in registers according to their order in memory. A variable-sized structure is always passed on the stack and any remaining argument registers go unused. If the size of a structure is not an integral number of words, padding is inserted at one end of the structure. For structures smaller than a word, the padding is always in the most-significant part of the word. A structure larger than a word is padded in the last bytes of the last argument word, so that the structure is contiguous when the registers are stored to consecutive words of memory.

Values of user-defined TIE types cannot be passed as arguments. (That is, they cannot be arguments of procedure calls; they may still be used as arguments of certain intrinsic functions and macros that do not correspond to real procedure calls.)

8.1.5 Return Values

Values of four words or less are returned in registers. The callee places the return value in registers beginning with AR[2] and continuing up to (and including) AR[5], depending on the size of the value. For a `CALL`*`N`* instruction (where *N* is 0 for the CALL0 ABI, or where *N* is 4, 8, or 12 for the windowed register ABI) the caller receives these values in registers AR[*N*+2] through AR[*N*+5]. (Note that, as with arguments, this limits the use of `CALL12` instructions. A `CALL12` instruction can only be used when the return value is two words or less; only AR[*N*+2] and AR[*N*+3] can be used when *N*=12.)

Return values smaller than a word are stored in the least-significant part of AR[2], with the upper bits set to zero for unsigned values or sign-extended for signed values.

Values larger than four words are returned by invisible reference. The caller passes a pointer as an invisible first argument and the callee stores the return value in the memory referenced by the pointer. The memory allocated by the caller must have the appropriate size and alignment for the return value.

Even though values of user-defined types cannot be passed as arguments, they are allowed as return values. If a procedure returns such a value, it is stored in the first register of the register file associated with that user-defined type.

8.1.6 Variable Arguments

Variable argument lists are handled in the same way as other arguments. There is no change to the calling convention for functions with variable argument lists.

8.1.7 Other Register Conventions

In addition to the general-purpose AR register file, Xtensa processors may contain a variety of other register files, special registers, and TIE states (which may be mapped to user registers). The conventions for saving and restoring these registers across function calls vary. Some are caller-saved, which means that a function does not need to save those registers to the stack before modifying them, because it can assume that the caller has already saved them. For callee-saved registers, the responsibility is reversed and the callee function must save the original values of the registers that it modifies. Some other registers are global — any changes to their values persist across function calls — and for some others, the usage conventions are not specified.

Unless otherwise specified, the default convention is that all registers are caller-saved. The exceptions are:

- When using the CALL0 ABI, several of the AR registers are callee-saved (see Table 8–243 on page 589).
- No convention is specified for the use of TIE states — the programmer can decide how to use TIE states. If you are using TIE states together with cooperative (non-preemptive) context switching, be careful that your use of TIE states matches the assumptions of the operating system. The operating system may assume that TIE states need not be saved when a context switch primitive is invoked; that is, it may assume that TIE states are caller-saved.
- The following special registers and user registers are global: `LITBASE`, `THREADPTR`, and `FCR`. These registers are used for special purposes and typically keep the same values across function calls.

As a consequence of the `LOOP` special registers (`LBEG`, `LEND`, and `LCOUNT`) being caller-saved, the `LOOP` instructions should not be used for loops containing function calls. Doing so would require saving and restoring the `LOOP` registers around the call, which would overwhelm the advantage of the `LOOP` instructions.

8.1.8 Nested Functions

Some languages (including C with a GCC extension) allow nested functions. A function A nested inside another function B must be able to access the local variables of both A and B. Implementing this requires that when B calls A, it must somehow pass to A information to allow locating B's stack frame. Some implementations of nested functions use a data structure known as a "display" for this purpose. GCC uses the simpler alternative of passing a "static chain" as an invisible argument to the nested function. The static chain is simply a pointer to the caller's stack frame. This approach is preferable to using a display as long as functions are not deeply nested.

Because nested functions may be called indirectly through pointers, the caller may not be able to detect when it is calling a nested function. Therefore, the invisible static chain argument must be passed in a reserved location where it does not interfere with the other arguments. For the CALL0 ABI, the static chain is passed in register a8. For the windowed register ABI, there are no registers available to hold the static chain, and the stack locations at positive offsets from SP are all used for passing normal arguments. The solution is to store the static chain on the stack at a negative offset from the caller's stack pointer. The first four words below SP are reserved as a register save area, so the static chain is passed in the fifth word below SP. That is, the caller places the static chain in memory at [SP–20], and the callee reads it from [SP + *FRAMESIZE* – 20] where *FRAMESIZE* is the size of the callee's stack frame.

When the address of a nested function is stored into a pointer, the compiler actually emits code to dynamically create a small piece of executable code known as a "trampoline", and the pointer is set to reference the trampoline. When an indirect call is made through the pointer, the trampoline code sets the value of the static chain and then transfers control to the nested function. The trampoline code is allocated on the stack — this implies that it must be possible to execute code stored in the region of memory holding the stack. For example, when using nested functions that have their addresses taken, the stack cannot be located in a separate data memory.

This positioning of the static chain for the windowed register ABI has an implication for exception handlers. If an exception occurs after the static chain has been written but before the `ENTRY` instruction in the callee, the contents of memory from [SP–20] through [SP–1] must be preserved by the handler. Because of the register overflow save area, the contents of memory from [SP–16] to [SP–1] must be preserved regardless, so the presence of the static chain simply adds one more word of memory that must be preserved.

8.1.9 Stack Initialization

Creating and initializing a stack for a new thread requires:

- reserving some memory,
- setting up the initial stack frame,
- setting the stack pointer to the initial frame, and
- setting the initial return address (in register `a0`) to zero.

If the initial procedure executed by the thread does not store any data in the initial stack frame, and if all the call instructions in the initial procedure use the CALL0 ABI or a window size of four, then the initial stack frame can be empty and requires no setup. The default C runtime initialization code meets these conditions, so that the stack can be initialized simply by setting the stack pointer to the high end of the reserved memory.

If the thread begins with some other code that may execute a `CALL8` or `CALL12` instruction or that requires storage on the stack, the initial frame must be constructed before jumping to the initial procedure. The size of the initial frame is equal to the sum of the local storage requirements and the extra save area. The stack pointer should be initialized to the high end of the reserved memory less the size of the initial frame. Furthermore, assuming the thread begins executing with only the current register window loaded, the base save area at (sp – 16) must be initialized as if it had been written by a window overflow. Specifically, the stack pointer value stored at (sp – 12) must be set to the high end of the reserved stack area plus 16 bytes. This allows subsequent window overflows to locate the extra save area in the initial stack frame.

The return address register (`a0`) for the first procedure on the stack must be explicitly set to zero. This is used to mark the top of the stack for use by stack unwinding code.

The following code is an example of how the stack may be initialized to allow `CALL8` (but not `CALL12`) in the initial thread:

```
movi      a0, 0
movi      sp, stackbase + stacksize - 16
addi      a4, sp, 32          // point 16 past extra save area
s32e      a4, sp, -12         // access to extra save area
call8     firstfunction
```

The following code is an example of how the stack may be initialized to allow `CALL12` and “loc” bytes of locals and parameters in the initial thread (loc is a multiple of 16):

```
movi      a0, 0
movi      sp, stackbase + stacksize - loc - 32
addi      a4, sp, loc + 48    // point 16 past extra save area
s32e      a4, sp, -12         // access to extra save area
call12    firstfunction
```

8.2 Other Conventions

This section describes the usage conventions other than the Xtensa application binary interface (ABI).

8.2.1 Break Instruction Operands

The `break` (24-bit) instruction has two immediate 4-bit operands, and the `break.n` (narrow, 16-bit) instruction has one immediate 4-bit operand. These operands (informally called "break codes" in this section) can be used to convey relevant information to the debug exception handler. Their exact meaning is a matter of convention. However, some of the tools and software (debuggers, OS ports, and so forth) used with Xtensa cores necessarily make use of the break instructions, so some conventions had to be established. The conventions that have been adopted are described in this section.

Half of the break codes are reserved for use by software provided by Tensilica and its partners, leaving the remaining half for "user-defined" purposes. Note that making use of user-defined break codes usually requires special OS or monitor support, or at least having control of the debug exception handler (or of the external OCD software when OCD mode is enabled). Break code allocations are described in Table 8–245.

Break codes have been allocated for a number of *planted breakpoints* (breakpoints that replace some arbitrary pre-existing instruction, usually under control of a debugger or related software, and usually temporarily) and *coded breakpoints* (breakpoints explicitly coded in the assembly source).

Planted breakpoints have a narrow (16-bit) and a wide (24-bit) version. Because 24-bit instructions exist in all Xtensa processors, instructions 24-bits or wider may be replaced with a 24-bit `BREAK` instruction. With the density option, the narrow version (`BREAK.N`) must generally be used when replacing an existing narrow instruction. Otherwise a wide break instruction would overwrite two sequential instructions, the second of which could be the (now corrupted) target of a branch. Note that without the density option, only the wide form of the break instruction can be used because the narrow version does not exist.

A number of coded breakpoints have been defined to provide a means of making various exceptions (that is, illegal instructions, load/store errors, and so forth) visible to the debugger, which does not otherwise see these types of exceptions through the debug exception vector. These breakpoints necessarily require support from the OS (or RTOS). They are typically invoked by the OS for those exceptions and interrupts that neither the OS nor the application handles, thus providing an opportunity for a debugger (if one is active) to catch the condition. If the OS has its own mechanism for handling unregistered exceptions and interrupts, the relevant coded breakpoint is normally invoked before this mechanism (there often is no well-defined "after"). Thus, it is very important that the debug exception handler treat the coded breakpoint as a no-op if no debugger is active, to let the OS follow its default course of action. By convention, any `break 1,x` instruction must be skipped and ignored if no debugger is active. If the debug exception handler (or OCD software if OCD mode is enabled) detects the presence of a debugger, it will transfer control to the debugger. Otherwise, it must immediately resume execution at the instruction following the break (which requires incrementing EPC[DEBUGLEVEL] by two for `break.n` or by three for `break`), in effect making the break a no-op.

Another essential requirement for `break 1,0` through `break 1,5` is that the OS invoke these coded breakpoints in exactly the same context (core state) as when the exception was entered (except, necessarily, for PC and EXCSAVE*n*). This allows the debugger to know the exact state of the core at the time the exception (or interrupt) occurred, without requiring any OS dependency. For example, when detecting an unhandled level-1 user exception, the OS has typically saved (in EXCSAVE1 and possibly memory) and modified only a few address registers; these registers must all be restored prior to executing the `break 1,1` instruction. The debug exception handler can then examine all registers as they were when the user exception occurred, including examining EXCCAUSE to determine which exception occurred, and so forth. Similarly, following a `break 1,2` it can resolve which interrupt occurred using EPS[DEBUGLEVEL].INTLEVEL.

Coded breakpoints can always use the wide (24-bit) form of the break instruction, so they were not allocated from the limited number of narrow break instructions.

Table 8–245. Breakpoint Instruction Operand Conventions

Breakpoint Instruction	Type	Description
`break 0,0`	planted	Breakpoints set by host debugger for debugging programs. These break instruction appear in code as a result of one of the following actions: ▪ The debugger can request the monitor to write the breakpoint instruction into the code. ▪ The debugger can explicitly write this instruction into the code.
`break 0,1`	planted	Breakpoints set by the monitor or OCD software for its own purposes. For example, xmon uses this breakpoint to detect and intercept UART interrupts. Ideally the presence of these breaks in the code is hidden from the debugger.
`break 0,2` to `0,15`	(undefined)	Reserved (Tensilica)
`break 1,0`	coded	Signals an unhandled level 1 kernel exception
`break 1,1`	coded	Signals an unhandled level 1 user exception
`break 1,2`	coded	Signals an unhandled high-priority interrupt
`break 1,3`	coded	Signals an unhandled window overflow or underflow exception (unlikely to be invoked)
`break 1,4`	coded	Signals an unhandled double exception
`break 1,5`	coded	Signals an unhandled memory error exception

Table 8–245. Breakpoint Instruction Operand Conventions (continued)

Breakpoint Instruction	Type	Description
`break 1,6` to `1,13`	coded	Reserved (Tensilica)
`break 1,14`	coded	Issue a request through the debugger. Any use of this break instruction is debugger-specific. For example, certain versions of GDB use this to implement target initiated host I/O.
`break 1,15`	coded	Transfer control to debugger if present. This is typically inserted manually in the code for debugging purposes, or to signal critical events that should cause entry into the debugger if one is active, but be ignored otherwise.
`break 2,x` to `7,x`	(undefined)	Reserved (Tensilica)
`break 8,x` to `15,x`	(undefined)	User-defined
`break.n 0`	planted	Same as `break 0,0`, but can also replace narrow (16-bit) instructions.
`break.n 1`	planted	Same as `break 0,1`, but can also replace narrow (16-bit) instructions.
`break.n 2` to `7`	(undefined)	Reserved (Tensilica)
`break.n 8` to `15`	(undefined)	User-defined

8.2.2 System Calls

The details of system calls are inherently dependent on the operating system, but there are a few conventions that apply to all systems. The `SYSCALL` instruction has no immediate operands, so the system call parameters are passed in registers. Each operating system is free to define its own register usage for system call parameters, with the exception that the system call request code must always be in register a2.

The system call request code 0 must be defined for all systems that use the windowed register ABI. (If the Xtensa processor configuration uses the CALL0 ABI, system call 0 need not be implemented.) The purpose of system call 0 is to flush the register windows to the stack. It is often useful to have a portable and reasonably efficient means of flushing register windows, such as when walking up the stack to find an exception handler. This system call provides an easy way to flush the register windows on all systems.

In general, each operating system can define its own conventions for which general-purpose registers may be modified by a system call, including which registers will hold any return values or error codes. For system call 0 in particular, no return value is expected and each operating system must guarantee that no general-purpose registers other than a2 will be modified. The value in a2 upon return from system call 0 depends on the operating system.

8.3 Assembly Code

This section describes various things of interest to the assembly language writer, including some examples.

8.3.1 Assembler Replacements and the Underscore Form

Machine code generated by the assembler may include opcode replacements for certain assembler opcodes. For example:

- The assembler can turn `ADD` into `ADD.N`, or `ADDI` into `ADDI.N`, and so forth when the density option is enabled.
- The assembler substitutes a different instruction when an operand is out of range. For example, it turns `MOVI` into `L32R` when the immediate is outside the range -2048 to 2047.
- By default, the assembler handles branches that won't reach. For example, writing:

```
        beq  a1, a2, label
```

might actually generate:

```
        bne  a1, a2, .L1
        j    label
.L1:
```

if `label` is too far to reach with a simple `beq` instruction.

These transformations can be disabled by prefixing the instruction name with an underscore (for example,`_ADD`) and with pseudo-ops. The assembler directives`.begin` and `.end` with `no-transform` can also be used to enable and disable these transformations. See the *GNU Assembler User's Guide* for more detail.

8.3.2 Instruction Idioms

Table 8–246 specifies the preferred instruction idioms for common operations. These idioms are specified using only core instructions; in some cases substituting density instructions would be appropriate.

Table 8–246. Instruction Idioms

Operation	Preferred Idiom
AR[x] ← AR[y]	or ax, ay, ay (generated by the MOV assembler macro) (or if present, use 16-bit option MOV.N)
AR[x] ← not AR[y]	movi at, -1 xor ax, ay, at
AR[x] ← AR[y] and not AR[z]	and at, ay, az xor ax, ay, at
AR[x] ← imm32	l32r ax, literalpooloffset
AR[x] ← AR[y] << AR[z]	ssl az sll ax, ay
AR[x] ← AR[y] $>>_u$ AR[z]	ssr az srl ax, ay
AR[x] ← AR[y] $>>_s$ AR[z]	ssr az sra ax, ay
AR[x] ← rot(AR[y], AR[z])	ssa az src ax, ay, ay
AR[x] ← byteswap(AR[y])	ssai 8 srli ax, ay, 16 src ax, ax, ay src ax, ax, ax src ax, ay, ax
if AR[x] ≤ AR[y] goto L	bge ay, ax, L
if AR[x] > AR[y] goto L	blt ay, ax, L
if AR[x] ≤ imm goto L	blti ax, imm+1, L
if AR[x] > imm goto L	bgei ax, imm+1, L
AR[x] ← AR[y] ≠ AR[z]	movi at, 1 xor ax, ay, az movnezax, at, ax
AR[x] ← AR[y] = AR[z]	movi ax, 1 bne ay, az, L movi ax, 0 L:
AR[x] ← AR[y] ≠ 0	movi at, 1 movi ax, 0 movnez ax, at, ay
AR[x] ← AR[y] = 0	movi at, 1 movi ax, 0 moveqz ax, at, ay

Table 8–246. Instruction Idioms (continued)

Operation	Preferred Idiom
64-bit add (x ← y + z)	add ax0, ay0, az0 add ax1, ay1, az1 bgeu ax0, az0, L1 addi ax1, ax1, 1 L1:
64-bit subtract (x ← y − z)	sub ax0, ay0, az0 sub ax1, ay1, az1 bgeu ay0, az0, L addi ax1, ax1, -1 L:
64-bit compare and branch if x < y goto L	blt ax1, ay1, L bne ax1, ay1, L1 bltu ax0, ay0, L L1:
64-bit multiply (x ← y × z)	mull ax0, ay0, az0 muluh ax1, ay0, az0 mull t, ay0, az1 add ax1, ax1, t mull t, ay1, az0 add ax1, ax1, t
BR[x] ← BR[y]	orb bx, by, by
BR[x] ← 0	xorb bx, b0, b0
BR[x] ← 1	orbc bx, b0, b0

8.3.3 Example: A FIR Filter with MAC16 Option

With the MAC16 Option, a portion of a real FIR filter might be:

```
input[next] = sample;// put sample into history array
acc = 0x4000;        // for rounding
for (i = 0; i < n; i += 1) {
     acc += input[i > next ? next-i+n : next-i] * coeff[i];
}
output[next] = acc >> 15;
next = next == N-1 ? 0 : next+1;
```

The read of the accumulator and shift is done as follows:

```
rsr  a6, acclo       // read 40-bit ACC
rsr  a7, acchi       // ...
ssai 15              // convert back to fractional 16
src  a2, a7, a6      // bit form
clampsa2, a2, 15     // clamp to 16 bits
```

To simplify the coding, change the preceding to store data in the input array backward so that the array references are all increments instead of decrements. Now convert it into two loops to avoid the circular addressing:

```
input[next] = in;
acc = 0x4000;
j = 0;
for (i = next; i < N; i += 1, j += 1) {
     acc += input[i] * coeff[j];
}
for (i = 0; i < next; i += 1, j += 1) {
     acc += input[i] * coeff[j];
}
next = next == 0 ? N-1 : next-1;
```

and then implement the loops with two calls to an assembler subroutine:

```
mac16_dot (N - next, &input[next], &coeff[0]);
mac16_dot (next, &input[0], &coeff[N - next]);
```

The MAC16 assembler for `mac16_dot` is:

```
// FIR Filter using MAC16

// Copyright 1999 Tensilica Inc.
// These coded instructions, statements, and computer programs are
// Confidential Proprietary Information of Tensilica Inc. and may not
be
// disclosed to third parties or copied in any form, in whole or in
part,
// without the prior written consent of Tensilica Inc.

// Exports
.global mac16_set_acc
.global mac16_acc
.global mac16_dot

// Use defines to make the code below less endian-specific
#if __XTENSA_EL__
#  define MULA00 mula.dd.ll
#  define MULA22 mula.dd.hh
#  define MULA02 mula.dd.lh
#  define MULA20 mula.dd.hl
#  define MULA00L mula.dd.ll.ldinc
#  define MULA22L mula.dd.hh.ldinc
#  define MULA02L mula.dd.lh.ldinc
#  define MULA20L mula.dd.hl.ldinc
#  define BBCI(_r,_b,_l) bbci _r, _b, _l
#  define BBSI(_r,_b,_l) bbsi _r, _b, _l
```

```
#endif
#if __XTENSA_EB__
#  define MULA00 mula.dd.hh
#  define MULA22 mula.dd.ll
#  define MULA02 mula.dd.hl
#  define MULA20 mula.dd.lh
#  define MULA00L mula.dd.hh.ldinc
#  define MULA22L mula.dd.ll.ldinc
#  define MULA02L mula.dd.hl.ldinc
#  define MULA20L mula.dd.lh.ldinc
#  define BBCI(_r,_b,_l) bbci _r, 31-(_b), _l
#  define BBSI(_r,_b,_l) bbsi _r, 31-(_b), _l
#endif


#include <machine/specreg.h>

     .text

// void mac16_set_acc(int hi, int lo)
     .align4
mac16_set_acc:
     entrysp, 16
     wsr  a2, ACCHI
     wsr  a3, ACCLO
     retw

// int mac16_acc(int shift)
     .align4
mac16_acc:
     entrysp, 16
     ssr  a2
     rsr  a2, ACCHI
     rsr  a3, ACCLO
     src  a2, a2, a3
     retw

// int mac16_dot (int n, int16* a, int16* b)
     .align4
mac16_dot:
     entrysp, 16
     // a2: n
     // a3: a[]
     // a4: b[]
     blti a2, 1, .sameret// if n <= 0, nothing to do
     addi a3, a3, -4// compensate for pre-increment
     addi a4, a4, -4// compensate for pre-increment
     xor  a5, a3, a4// check if vectors have same alignment
     BBSI(a5, 1, .diffalign)
```

```
.samealign:// vectors have same alignment
     BBCI(a3, 1, .samewordalign)
     ldincm0, a3    // a[0]
     addi a3, a3, -2// undo overincrement, leave *a word-aligned
     ldincm2, a4    // b[0]
     addi a4, a4, -2// undo overincrement, leave *b word-aligned
     MULA22m0, m2   // add product of misaligned first values
     addi a2, a2, -1// finished one iteration
.samewordalign:// a[0] is word-aligned, b[0] is word-aligned
     srli a5, a2, 2 // will do 4 MACs per inner loop iteration
     beqz a5, .samemod4check// not even wind-up or wind-down
     addi a5, a5, -1// (n/4)-1 inner loop iterations
                    // (1 iteration done in wind-up/wind-down)
     // wind up
     ldincm0, a3    // m0 = a[1]:a[0]
     ldincm2, a4    // m2 = b[1]:b[0]
     ldincm1, a3    // m1 = a[3]:a[2]
     MULA00Lm3, a4, m0, m2// m3 = b[3]:b[2]; acc += a[0]*b[0]
     loopneza5, .sameloopend
.sameloop:// for i = 4; i < N-3; i += 4
     MULA22Lm0, a3, m0, m2// m0 = a[i+1]:a[i+0]; acc += a[i-4+1]:b[i-
4+1]
     MULA00Lm2, a4, m1, m3// m2 = b[i+1]:b[i+0]; acc += a[i-4+2]:b[i-
4+2]
     MULA22Lm1, a3, m1, m3// m1 = a[i+3]:a[i+2]; acc += a[i-4+3]:b[i-
4+3]
     MULA00Lm3, a4, m0, m2// m3 = b[i+3]:b[i+2]; acc += a[i+0]*b[i+0]
.sameloopend:
     // wind down
     MULA22m0, m2   // acc += a[i+1]*b[i+1]
     MULA00m1, m3   // acc += a[i+2]*b[i+2]
     MULA22m1, m3   // acc += a[i+3]*b[i+3]
.samemod4check:
     BBCI(a2, 1, .samemod2check)
     // count is 2 mod 4
     ldincm0, a3    // m0 = a[i+5]:a[i+4]
     ldincm2, a4    // m2 = b[i+5]:b[i+5]
     MULA00m0, m2   // acc += a[i+4]*b[i+4]
     MULA22m0, m2   // acc += a[i+5]*b[i+5]
.samemod2check:
     BBCI(a2, 0, .sameret)
     // count is 1 mod 2
     ldincm0, a3    // m0 = a[i+7]:a[i+6]
     ldincm2, a4    // m2 = b[i+7]:b[i+6]
     MULA00m0, m2   // acc += a[i+6]*b[i+6]
.sameret:
     retw

.diffalign:// vectors have different alignment
```

```
     BBCI(a3, 1, .diffwordalign)
     // a[0] is misaligned, b[0] is aligned
     ldincm0, a3    // a[0]
     addi a3, a3, -2// undo overincrement, leave *a word-aligned
     ldincm2, a4    // b[0]
     addi a4, a4, -2// undo overincrement, leave *b misaligned
     MULA20m0, m2   // add product of first values
     addi a2, a2, -1// finished one iteration
.diffwordalign:     // a[0] is now aligned, b[0] is misaligned
     srli a5, a2, 2 // will do 4 MACs per inner loop iteration
     ldincm3, a4    // m3 = b[0]:b[-1]
     beqz a5, .diffmod4check// not even wind-up or wind-down
     addi a5, a5, -1// (n/4)-1 inner loop iterations
                    // (1 iteration done in wind-up/wind-down)
     // wind up
     ldincm0, a3    // m0 = a[1]:a[0]
     ldincm2, a4    // m2 = b[2]:b[1]
     MULA02Lm1, a3, m0, m3// m1 = a[3]:a[2]; acc += a[0] * b[0]
     MULA20Lm3, a4, m0, m2// m3 = b[4]:b[3]; acc += a[1] * b[1]
     loopneza5, .diffloopend
.diffloop:// for i = 4; i < N-3; i += 4
     MULA02Lm0, a3, m1, m2// m0 = a[i+1]:a[i+0]; acc += a[i-4+2]*b[i-
4+2]
     MULA20Lm2, a4, m1, m3// m2 = b[i+2]:b[i+1]; acc += a[i-4+3]*b[i-
4+3]
     MULA02Lm1, a3, m0, m3// m1 = a[i+3]:a[i+2]; acc += a[i+0]*b[i+0]
     MULA20Lm3, a4, m0, m2// m3 = b[i+4]:b[i+3]; acc += a[i+1]*b[i+1]
.diffloopend:
     // wind down
     MULA02m1, m2   // acc += a[i+2] * b[i+2]
     MULA20m1, m3   // acc += a[i+3] * b[i+3]
.diffmod4check:
     BBCI(a2, 1, .diffmod2check)
     // count is 2 mod 4
     ldincm0, a3    // m0 = a[i+5]:a[i+4]
     MULA02m0, m3   // acc += a[i+4] * b[i+4]
     ldincm3, a4    // m3 = b[i+6]:b[i+5]
     MULA20m0, m3   // acc += a[i+5] * b[i+5]
.diffmod2check:
     BBCI(a2, 0, .diffret)
     // count is 1 mod 2
     ldincm0, a3    // m0 = a[i+7]:a[i+6]
     MULA02m0, m3   // acc += a[i+6] * b[i+6]
.diffret:
     retw
```

8.4 Performance

This book describes the Xtensa Instruction Set Architecture (ISA) but is not the reference for performance. The ISA is defined independently of its various implementations, so that software that targets the ISA will run on any its implementations. The ISA includes features that are not required by some of its implementations, but which will be important to include in software written today if it is to work on future implementations (for example, using `MEMW`, `EXTW`, and `EXCW`). While correct software must adhere to the ISA and not to the specifics of any of its implementations, it is sometimes important to know the details of an implementation for performance reasons, such as scheduling instructions to avoid pipeline delays. This chapter provides an overview of performance modeling.

8.4.1 Processor Performance Terminology and Modeling

It is important to have a model of processor performance for both code generation and simulation. However, the interactions of multiple instructions in a processor pipeline can be complex. It is common to simplify and describe pipeline and cache performance separately even though they may interact, because the information is used in different stages of compilation or coding. We adopt this approach, and then separately describe some of the interactions. It is also common to describe the pipelining of instructions with *latency* (the time an instruction takes to produce its result after it receives its inputs) and *throughput* (the time an instruction delays other instructions independent of operand dependencies) numbers, but this cannot accommodate some situations. Therefore, we adopt a slightly more complicated, but more accurate model. This model focuses on predicting when one instruction *issues* relative to other instructions. An instruction issues when all of its data inputs are available and all the necessary hardware functional units are available for it. Issue is the point at which computation of the instruction's results begins.

Instead of using a per-instruction latency number, instructions are modeled as taking their operands in various pipeline stage numbers, and producing results in various pipeline stage numbers. When instruction IA writes (or defines) X (either an explicit operand or implicit state register) and instruction IB reads (or uses) X, then instruction IB depends on IA.[1] If instruction IA defines X in stage SA (at the end of the stage), and instruction IB uses X in stage SB (at the beginning of the stage), then instruction IB can issue no earlier than $D = \max(SA - SB + 1, 0)$ cycles after IA issued. This is illustrated in Figure 8–54. If the processor reaches IB earlier than D cycles after IA, it generally delays IB's issue into the pipeline until D cycles have elapsed. When the processor delays an instruction because of a pipeline interaction, it is called an "interlock." For a few special dependencies (primarily those involving the special registers controlling exceptions,

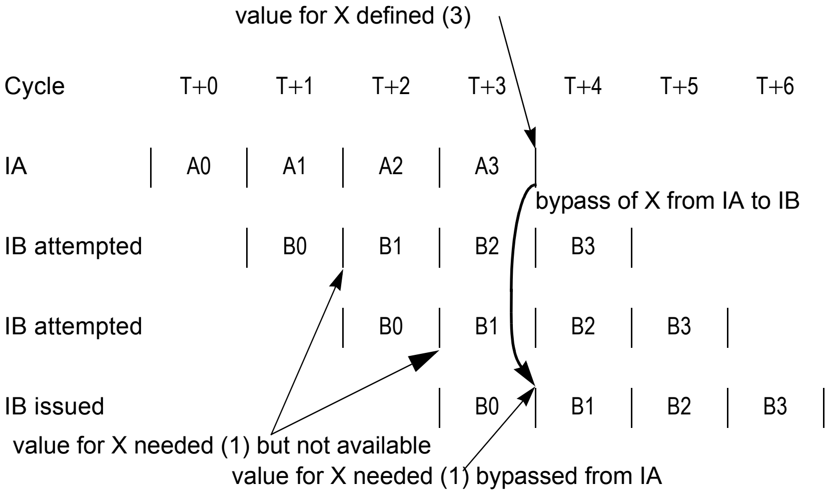
[1] This situation is called a "read after write" dependency. Other possible operand dependencies familiar to coders are "write after write" and "write after read."

interrupts, and memory management) the processor does not interlock. These situations are called “hazards.” For correct operation, code generation must insert `xSYNC` instructions to avoid hazards by delaying the dependent instruction. The `xSYNC` series of instructions is designed to accomplish this delay in an implementation-independent manner.

When an instruction is described as making one of its values available at the end of some stage, this refers to when the computation is complete, and not necessarily the time that the actual processor state is written. It is usual to delay the state write until at least the point at which the instruction is committed (that is, cannot be aborted by its own or an earlier instruction’s exception). In some implementations the state write is delayed still further to satisfy resource constraints. However, the delay in writing the actual processor state is usually invisible; most processors will detect the use of an operand that has been produced by one instruction and is being used by another even though the processor state has not been written, and forward the required value from one pipeline stage to the other. This operation is called *bypass*.

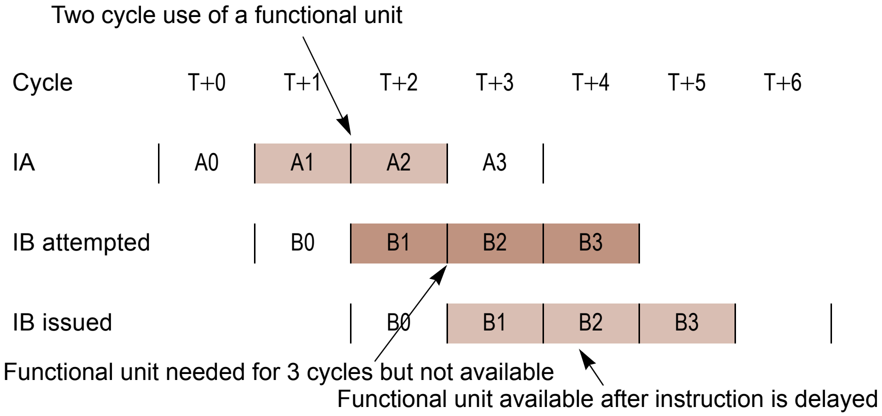
Instructions may be delayed in a pipeline for reasons other than operand dependencies. The most common situation is for two or more instructions to require a particular piece of the processor’s hardware (called a “functional unit”) to execute. If there are fewer copies of the unit than instructions that need to use the unit in a given cycle, the processor must delay some of the instructions to prevent the instructions from interfering with each other. For example, a processor may have only one read port for its data cache. If instruction IC uses this read port in its stage 4 and instruction ID uses the read port in its stage 3, then it would not be possible to issue IC in cycle 10 and ID in cycle 11, because they would both need to use the data cache read port in cycle 14. Typically, the processor would delay ID’s issue into the pipeline by one cycle to avoid conflict with IC.

Modern processor pipeline design tends to avoid the use of functional units in varying pipeline stages by different instructions and to fully pipeline functional unit logic. This means that most instructions would conflict with each other on a shared functional unit only if they issued in the same cycle. However, there are usually still a small number of cases in which a functional unit is used for several cycles. For example, floating-point or integer division may iterate for several cycles in a single piece of hardware. In this case, once a divide has started, it is not possible to start another divide until the first has left the iterative hardware. This is illustrated in Figure 8–55.



IA issues in cycle T+0, IB issues in cycle T+0+max(3–1+1,0) = T+3

Figure 8–54. Instruction Operand Dependency Interlock


Two cycle use of a functional unit
Cycle
T+0
T+1
T+2
T+3
T+4
T+5
T+6
IA
A0
A1
A2
A3
IB attempted
B0
B1
B2
B3
IB issued
B0
B1
B2
B3
Functional unit needed for 3 cycles but not available
Functional unit available after instruction is delayed

IA issues at T+0 and reserves the functional unit in cycles T+1 and T+2
IB tries to issue at T+1 and reserve the unit in T+2..T+4, but is blocked by IA’s T+2 reservation
IB is retried and issues in cycle T+2, thereby avoiding IA’s reservations

Figure 8–55. Functional Unit Interlock

8.4.2 Xtensa Processor Family

Many implementations of the Xtensa processor use a 5-stage pipeline capable of executing at most one instruction per cycle. The pipeline stages are described in Table 8–247. The first stage, I, is partially decoupled from the next, R, and R is partially decoupled from the last three stages, E, M, and W, which operate in lock-step. If an interlock condition is detected in the R stage, then in the next cycle the instruction is retried in R and a no-op is sent on to the E stage. If an instruction is held in R, then the word fetched in I is captured in a buffer.

Table 8–247. Xtensa Pipeline

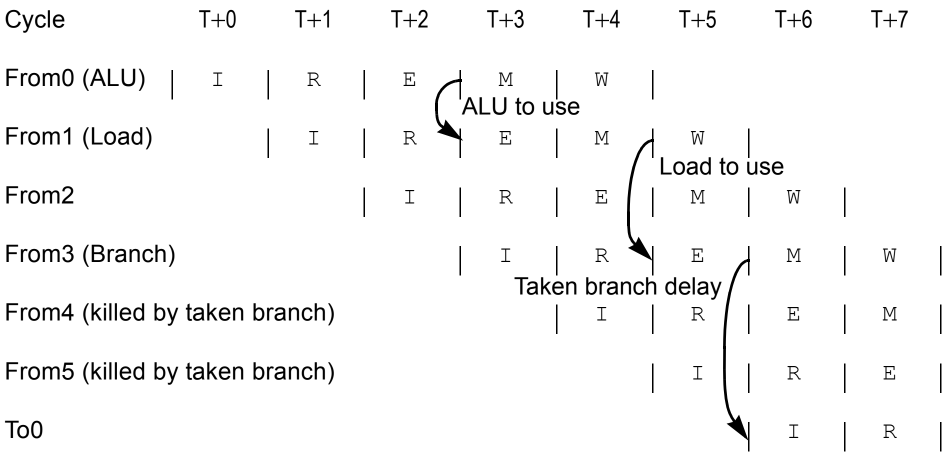
Name	Description
I	Instruction cache/RAM/ROM access Instruction cache tag comparison Instruction alignment
R	AR register file read Instruction decode, interlocking, and bypass Instruction cache miss recognition
E	Execution of most ALU-type instructions (ADD, SUB, etc.) Virtual address generation for load and store instructions Branch decision and address selection
M	Data cache/RAM/ROM access for load and store instructions Data cache tag comparison Data cache miss recognition Load data alignment
W	State writes (e.g. AR register file write)

The three primary implications of the Xtensa pipeline are shown in Figure 8–56.

- Instructions that depend on an ALU result can execute with no delay because their result is available at the end of E and is needed at the beginning of E by the dependent instruction.
- Instructions that depend on load instruction results must issue two cycles after the load because the load result is available at the end of its M stage and is needed at the beginning of E by the dependent instruction. For best performance, code generation should put an independent instruction in between the load and any instruction that uses the load result.
- Finally, the branch decision occurs in E, and for taken branches must affect the I stage of the target fetch, and so there are two fetched fall-through instructions that are killed on taken branches.

The base processor uses 32-bit aligned fetches from the instruction cache/RAM/ROM. Processors with instructions larger than 32 bits in size use fetches big enough to fetch at least one instruction per cycle. If the target of a branch is an instruction that crosses a fetch boundary, then two fetches will be required before the entire instruction is available, and so the target instruction will begin three cycles after the branch instead of two. For best performance, code generation should align 24-bit targets of frequently taken branches on 0 or 1 mod 4 byte boundaries, and 16-bit targets on 0, 1, or 2 mod 4 byte boundaries.

The processor avoids overflowing its write buffer by interlocking in the R stage on stores when the write buffer is full or might become full from stores in the E and M stages.



Cycle	T+0	T+1	T+2	T+3	T+4	T+5	T+6	T+7
From0 (ALU)	I	R	E	M	W			
From1 (Load)		I	R	E	M	W		
From2			I	R	E	M	W	
From3 (Branch)				I	R	E	M	W
From4 (killed by taken branch)					I	R	E	M
From5 (killed by taken branch)						I	R	E
To0							I	R

Figure 8–56. Xtensa Pipeline Effects

Refer to a specific Xtensa processor data book for detailed descriptions of processor performance and tables of pipeline stages where operands are used and defined.

A. Differences Between Old and Current Hardware

A.1 Added Instructions

Instructions have been added to the instruction set at various points. Most have been added as a part of new options, but a few have been added to existing options. Table 9-248 shows instructions added to existing options along with the first implementation in which they were added.

Table 9-248. Instructions Added

Instruction	First Implementation Containing the Instruction
`DIWB`	T1050
`DIWBI`	T1050
`EXTW`	RA-2004.1
`NOP` (actual instruction rather than assembly macro)	RA-2004.1
`RER`	RC-2009.0
`WER`	RC-2009.0
`XSR`	T1040

A.2 Xtensa Exception Architecture 1

As is described in Section 4.4.1, there are two variants of the Exception Option. Xtensa Exception Architecture 1 (XEA1) is no longer available for new hardware and this section describes the differences between it and Xtensa Exception Architecture 2 (XEA2), which is described in the option chapter in Section 4.4.1.

The biggest difference between the two is that where XEA2 has a bit, `PS.EXCM`, that causes certain effects in the hardware that are useful on entering and leaving exceptions and interrupts, XEA1 has that functionality bundled into the setting of the `PS.INTLEVEL` field. There is no provision for either ring protection or double exceptions in XEA1.

The following subsections describe the differences in more detail.

A.2.1 Differences in the PS Register

The following fields of the PS register (see page 87) are different in XEA1:

- There is no PS.EXCM field in XEA1
- There is no PS.RING field in XEA1
- PS.INTLEVEL always exists in XEA1 (added by the Exception Option) instead of appearing with the Interrupt Option. In this case CINTLEVEL is 0 for normal operation and 1 when executing in an exception handler.

Some of the functions surrounding the fields of the PS register are also different from later behavior (see Section 4.4.1.3). In XEA1:

- CEXCM ← PS.INTLEVEL ≠ 0
- CRING ← 0
- CINTLEVEL ← PS.INTLEVEL
- CWOE ← PS.WOE
- CLOOPENABLE ← 1

In XEA1, there is no architectural provision to take an instruction related exception when CINTLEVEL is greater than zero, but in actual hardware delivered it was possible to do under carefully controlled situations.

In XEA1, the PS register is reset to the value $0^{28}||1^{4}$, which is different from what is given in Section 3.6 for XEA2.

A.2.2 Exception Semantics

Instead of the semantics shown in Section 4.4.1.10, exceptions have the following semantics in Xtensa Exception Architecture 1 (XEA1):

```
procedure Exception(cause)
     EPC[1] ← PC
     PS.INTLEVEL ← 1
     n ← if $\text{WindowStart}_{\text{WindowBase}+1}$ then 2'b01
          else if $\text{WindowStart}_{\text{WindowBase}+2}$ then 2'b10
          else if $\text{WindowStart}_{\text{WindowBase}+3}$ then 2'b11
          else 2'b00
     if PS.UM then
          EXCCAUSE ← cause
          nextPC ← UserExceptionVector
          PS.UM ← 0
          PS.WOE ← 0
     elseif n ≠ 2'b00 then
          PS.OWB ← WindowBase
          PS.WOE ← 0
```

```
            m ← WindowBase + (2'b00||n)
            nextPC ← if WindowStart$_{m+1}$ then WindowOverflow4
                 else if WindowStart$_{m+2}$ then WindowOverflow8
                 else WindowOverflow12
            WindowBase ← m
       else
            EXCCAUSE ← cause
            nextPC ← KernelExceptionVector
            -- note PS.WOE left unchanged
            -- note PS.UM is already 0
       endif
  endprocedure Exception
```

The intent of the window checks in Xtensa Exception Architecture 1 is to allow the kernel exception handler to use CALLX12 without taking an exception. This allows the handler to “save” 12 registers using the windowed-register mechanism instead of using 12 loads and 12 stores. This results in low-overhead kernel exceptions. When the window overflow exception is invoked instead of the requested exception, the RFWO from the handler will attempt to re-execute the instruction that caused the original exception, and this time the kernel exception handler will be invoked. This feature has proved difficult to use in operating systems.

User vector mode exceptions work differently because it is usually necessary to switch stacks when going from the program stack to the exception stack, and this involves storing all windows to the program stack.

Instead of the semantics shown in Section 4.7.1.3, window checks have the following semantics in Xtensa Exception Architecture 1 (XEA1):

```
  procedure WindowCheck (wr, ws, wt)
       n ← if (wr ≠ 2'b00 or ws ≠ 2'b00 or wt ≠ 2'b00)
                 and WindowStart$_{WindowBase}$+$_{1}$ then 2’b01
            else if (wr$_{1}$ or ws$_{1}$ or wt$_{1}$)
                 and WindowStart$_{WindowBase}$+$_{2}$ then 2’b10
            else if (wr = 2'b11 or ws = 2'b11 or wt = 2'b11)
                 and WindowStart$_{WindowBase}$+$_{3}$ then 2’b11
            else 2’b00
       if CWOE = 1 and n ≠ 2’b00 then
            PS.OWB ← WindowBase
            m ← WindowBase + (2'b00||n)
            PS.WOE ← 0
            PS.INTLEVEL ← 1
            EPC[1] ← PC
            nextPC ← if WindowStart$_{m+1}$ then WindowOverflow4
                 else if WindowStart$_{m+2}$ then WindowOverflow8
                 else WindowOverflow12
            WindowBase ← m
       endif
  endprocedure WindowCheck
```

A.2.3 Checking ICOUNT

The procedure for taking an ICOUNT interrupt is different from the one given in Section 4.7.6.8. Instead of setting PS.EXCM, it clears PS.WOE and PS.UM as shown here:

```
procedure checkIcount ()
    if CINTLEVEL < ICOUNTLEVEL then
        if ICOUNT ≠ -1 then
            ICOUNT ← ICOUNT + 1
        elseif CINTLEVEL < DEBUGLEVEL then
            EPC[DEBUGLEVEL] ← PC
            EPS[DEBUGLEVEL] ← PS
            DEBUGCAUSE ← 1
            PC ← InterruptVector[DEBUGLEVEL]
            PS.WOE ← 0
            PS.UM ← 0
            PS.INTLEVEL ← DEBUGLEVEL
        endif
    endif
endprocedure checkIcount
```

A.2.4 The BREAK and BREAK.N Instructions

In XEA1 the BREAK and BREAK.N instructions do not affect PS.EXCM, since it does not exist, but set PS.UM ← 0 and PS.WOE ← 0 instead.

A.2.5 The RETW and RETW.N Instructions

In XEA1 the RETW and RETW.N instructions are not affected by and do not affect PS.EXCM, since it does not exist. In the underflow case, before setting EPC[1] ← PC, these instructions set PS.WOE ← 0 and PS.INTLEVEL ← 1 instead.

A.2.6 The RFDE Instruction

There is no RFDE instruction in XEA1.

A.2.7 The RFE Instruction

In XEA1 the RFE instruction does not affect PS.EXCM, since it does not exist, but sets PS.INTLEVEL ← 0 instead. In XEA1, it is used only to return from exceptions that went to the kernel exception vector.

A.2.8 The RFUE Instruction

XEA1 supports the RFUE instruction, which is nearly identical to the RFE instruction but sets PS.UM ← 1 and PS.WOE ← 1 in addition. A partial description is given in Chapter 6, page 243. The following instruction entry shows the RFUE instruction that is not fully described in Chapter 6. Note that an ESYNC instruction needs to be used between a WSR/XSR.EPC1 and an RFUE instruction.

Instruction Word

23																							0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0

24

Required Configuration Option:

Exception Option (Xtensa Exception Architecture 1 Only)

Assembler Syntax

```
RFUE
```

Description

RFUE exists only in Xtensa Exception Architecture 1. It is an illegal instruction in Xtensa Exception Architecture 2 and above.

RFUE returns from an exception that went to the UserExceptionVector (that is, a non-window synchronous exception or level-1 interrupt that occurred while the processor was executing with PS.UM set). It sets PS.UM back to 1, clears PS.INTLEVEL back to 0, sets PS.WOE back to 1, and then jumps to the address in EPC[1].

RFUE is a privileged instruction.

Operation

```
if CRING ≠ 0 then
     Exception (PrivilegedInstructionCause)
else
     PS.UM ← 1
     PS.INTLEVEL ← 0
     PS.WOE ← 1
     nextPC ← EPC[1]
endif
```

Exceptions

- EveryInst Group (see page 244)
- GenExcep(IllegalInstructionCause) if Exception Option

A.2.9 The RFWO and RFWU Instructions

In XEA1 the RFWO and RFWU instructions do not affect PS.EXCM, since it does not exist, but set PS.INTLEVEL ← 0 and PS.WOE ← 1 instead.

A.2.10 Exception Virtual Address Register

The exception virtual address register, EXCVADDR, does not exist in XEA1. There are no memory management tables to refill and so it is not absolutely necessary. On other memory exceptions, system software must decode the instruction to determine the memory address involved if it wishes to know.

A.2.11 Double Exceptions

There is never a DEPC register in XEA1. Double exceptions are not generally recoverable in XEA1 and often not detectable.

A.2.12 Use of the RSIL Instruction

The RSIL instruction is typically used for executing a region of code at a new level:

```
RSIL      a2, newlevel
code to be executed at newlevel
WSR       a2, PS
```

In XEA2, the atomicity of the RSIL instruction is a convenience, but in XEA1 it is required to avoid race conditions that have to do with the fact that returning from exceptions sets PS.INTLEVEL to zero.

A.2.13 Writeback Cache

No writeback data cache is available in XEA1.

A.2.14 The Cache Attribute Register

In XEA1, the Options for Memory Protection and Translation in Section 4.6 and the corresponding TLB management instructions are not available. Instead, functionality similar to the Region Protection Option described in Section 4.6.3 is available through the cache attribute register. Table 9-249 shows the cache attribute register and its addition as a Special Register.

Table 9-249. Cache Attribute Register

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number[1]
CACHEATTR	1	32	Cache attribute	R/W	98

[1] Registers with a Special Register assignment are read and/or written with the RSR, WSR, and XSR instructions. See Table 5–127 on page 205.

The following table shows the Cache Attribute Special Register as it is implemented in XEA1 and described as current Special Registers are described in Chapter 5.

Table 9-250. Cache Attribute Special Register

<table>
<tr><th>SR#</th><th>Name</th><th colspan="3">Description</th><th>Reset Value</th></tr>
<tr><td>98</td><td>CACHEATTR</td><td colspan="3">Cache Attribute Register</td><td>32’h22222222</td></tr>
<tr><th colspan="2">Option</th><th>Count</th><th>Bits</th><th>Privileged?</th><th>XSR Legal?</th></tr>
<tr><td colspan="2">Exception Option Architecture 1</td><td>1</td><td>32</td><td></td><td>Yes</td></tr>
<tr><th colspan="3">WSR Function</th><th colspan="3">RSR Function</th></tr>
<tr><td colspan="3">CACHEATTR ← AR[t]</td><td colspan="3">AR[t] ← CACHEATTR</td></tr>
<tr><th colspan="3">Other Changes to the Register</th><th colspan="3">Other Effects of the Register</th></tr>
<tr><td colspan="3"></td><td colspan="3">Any instruction/data address translation</td></tr>
<tr><th colspan="6">Instruction ⇒ xSYNC ⇒ Instruction</th></tr>
<tr><td colspan="6">WSR/XSR CACHEATTR ⇒ ESYNC ⇒ RSR/XSR CACHEATTR</td></tr>
<tr><td colspan="6">WSR/XSR CACHEATTR ⇒ ISYNC ⇒ Any Instruction address translation that depends on new value</td></tr>
<tr><td colspan="6">WSR/XSR CACHEATTR ⇒ DSYNC ⇒ Any data address translation that depends on the change</td></tr>
</table>

The single register controls protection for all of memory and for both instruction and data fetches. As shown in Figure 9-57, the register consists of eight 4-bit attribute fields. For any memory access, one of the attrn (attribute) fields is chosen for both instruction and data accesses by the following algorithm:

```
b ← vAddr31..29
cacheattr ← CACHEATTR(b||2'b11)..(b||2'b00)
```

This allows the cache attributes to be separately specified for each 512MB of address space, just as with the attributes in the Region Protection Option described in Section 4.6.3. And as with that option, no translation of addresses is done.

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
attr7	attr6	attr5	attr4	attr3	attr2	attr1	attr0
4	4	4	4	4	4	4	4

Figure 9-57. CACHEATTR Register

The resulting attribute is interpreted for both cache and local memory accesses as described in Section 4.6.3.3, except that writeback caches are not available. It is in this sense that the Region Protection Option is upward compatible with XEA1.

After changing the attribute of a region by `WSR` to `CACHEATTR`, the operation of instruction fetch from that region is undefined until an `ISYNC` instruction is executed. Thus software should not change the cache attribute of the region containing the current `PC`.

After changing the attribute of a region by `WSR` to `CACHEATTR`, the operation of loads from and stores to that region are undefined until a `DSYNC` instruction is executed.

The processor sets every region of `CACHEATTR` to bypass (`4'b0010`) on processor reset.

The following pseudocode describes the accessing of the `CACHEATTR` register.

```
function fcadecode (ca)-- cacheattr decode for fetch
     if not (ca = 4'd1 or ca = 4'd2 or ca = 4'd3 or ca = 4'd4) then
          fcadecode ← undefined^8||1
     else
          usehit ← ca = 4'd1 or ca = 4'd3 or ca = 4'd4
          allocate ← ca = 4'd1 or ca = 4'd3 or ca = 4'd4
          writethru ← undefined
          isolate ← undefined
          guard ← 0
          coherent ← 0
          prefetch ← 0
          streaming ← 0
          fcadecode ← streaming||prefetch||coherent||guard
                   ||isolate||writethru||allocate||usehit||0
     endif
endfunction fcadecode

function lcadecode (ca)-- cacheattr decode for load
     if ca > 4'd4 and ca ≠ 4'd14 then
          lcadecode ← undefined^8||1
     else
          usehit ← ca ≠ 4'd2
          allocate ← ca = 4'd1 or ca = 4'd3 or ca = 4'd4
          writethru ← undefined
          isolate ← ca = 4'd14
          guard ← 0
          coherent ← 0
          prefetch ← 0
          streaming ← 0
          lcadecode ← streaming||prefetch||coherent||guard
                    ||isolate||writethru||allocate||usehit||0
     endif
endfunction lcadecode

function scadecode (ca)-- cacheattr decode for store
     if ca > 4'd4 and ca ≠ 4'd14 then
          scadecode ← undefined$^8$||1
     else
          usehit ← undefined
          allocate ← ca = 4'd3 or ca = 4'd4
          writethru ← ca < 4'd4
          isolate ← ca = 4'd14
          guard ← 0
          coherent ← 0
          prefetch ← 0
          streaming ← 0
          scadecode ← streaming||prefetch||coherent||guard
                    ||isolate||writethru||allocate||usehit||0
     endif
endfunction scadecode
```

A.3 New Exception Cause Values

Beginning with the RB-2006.0 release, the `EXCCAUSE` register, as indicated in Table 4–64 on page 89, can, in limited cases have different values than it did before that. In particular, exceptions which used to result in `EXCCAUSE` code 2 (`InstructionFetchErrorCause`) are now split into three values. `EXCCAUSE` code 2 (`InstructionFetchErrorCause`) now covers only those errors occuring inside the Xtensa processor. `EXCCAUSE` code 12 (`InstrPIFDataErrorCause`) now covers data errors on the PIF for Instruction fetch and `EXCCAUSE` code 14 (`InstrPIFAddrErrorCause`) now covers address errors on the PIF for Instruction fetch. Similarly, exceptions which used to result in `EXCCAUSE` code 3 (`LoadStoreErrorCause`) are now split into three values. `EXCCAUSE` code 3 (`LoadStoreErrorCause`) now covers only those errors occuring inside the Xtensa processor. `EXCCAUSE` code 13 (`LoadStorePIFDataErrorCause`) now covers data errors on the PIF for Load/Store and `EXCCAUSE` code 15 (`LoadStorePIFAddrErrorCause`) now covers address errors on the PIF for Load/Store.

This change was made to make it easier to separate errors caused by the system from errors caused by the Xtensa processor itself during debugging. If exception code is upgraded so that exceptions with EXCCAUSE set to values 12-15 are routed to the code that handled EXCCAUSE 2 and 3 as appropriate, then the previous functionality is retained.

A.4 ICOUNTLEVEL

The ICOUNTLEVEL Special Register is undefined after reset instead of 4’hF, beginning with the RA-2004.1 release. This change should not cause any difficulty as the behavior is the same after reset since PS.INTLEVEL is 4’hF.

A.5 MMU Option Memory Attributes

As described in Section 4.6.5.10, T1050 used different MMU Option Memory Attributes. System software may use the subset of attributes (1, 3, 5, 7, 12, 13, and 14) that have not changed to support all Xtensa processors.

The specific differences for T1050 were:

- In Table 4–109 on page 178, rows with Attribute 0, 2, 4, 6, 8, and 10 were equivalent to the row with Attribute 12 in the table.
- In Table 4–109 on page 178, the row with Attribute 15 was equivalent to the row with Attribute 7, for the Data MMU but to the row with Attribute 12 for the Instruction MMU.
- In Table 4–109 on page 178 for Data Loads when writeback caches are not present, rows with Attributes 9 and 11 were called “No Allocate” instead of “Cached” and the column labeled “Fill Load”) contained “no” for instead of “yes”.

A.6 Special Register Read and Write

Before the RA-2004.1 release, Special Registers were read and written with the RSR, WSR, and XSR instructions. Each of these instructions takes one operand to indicate the Special Register that was the source or destination of the instruction, and another operand to indicate the AR register used as the other operand.

Beginning with the RA-2004.1 release, this trio of instructions was replaced with an individual trio of instructions for each Special Register. For example, the new instructions for accessing the LBEG register are called RSR.LBEG, WSR.LBEG, and XSR.LBEG. The new instructions take only one operand, which is the AR register. The old version of the instructions continues to be supported as an assembler macro that translates to the new ones.

The old trio of instructions was legal whether or not the Special Register accessed was defined in the particular implementation and, therefore, never produced an illegal instruction exception. Each of the new, much larger set of instructions is associated with a particular Special Register, and therefore is legal only if the associated register is defined. Each of the trio of instructions for an undefined register raises an illegal instruction exception when execution is attempted.

Rather than list several hundred individual instructions, Chapter 6 lists the instructions as RSR.*, WSR.*, and XSR.* and references the list of Special Registers in Chapter 5.

A.7 MMU Modification

In the RC.2009.0 release and after, the IVARWAY56 and DVARWAY56 parameters in Table 4–105 on page 159 must both be "Variable" whereas before that they must both be "Fixed". The functional operation of the MMU with the parameters set to Fixed may be emulated when the parameters are set to Variable. In other words, the function of the earlier MMU can be emulated by the later one.

A.8 Reduction of SYNC Instruction Requirements

For the T1050 release and releases before it, there were additional SYNC instruction requirements not listed in Section 5.3 on page 208. These additional SYNC instruction requirements are listed in Table 9–251, by subsection and in the same format used in Section 5.3. If these SYNC instructions are inserted in later releases where they are not needed, the code will still function correctly.

Table 9–251. T1050 Additional SYNC Requirements

Instruction	⇒	**xSYNC**	⇒	Instruction
		Section 5.3.2 on page 212		
WSR/XSR LBEG	⇒	ESYNC	⇒	RSR/XSR LBEG
WSR/XSR LEND	⇒	ESYNC	⇒	RSR/XSR LEND
		Section 5.3.3 on page 213		
WSR/XSR ACCLO	⇒	ESYNC	⇒	RSR/XSR ACCLO
WSR/XSR ACCHI	⇒	ESYNC	⇒	RSR/XSR ACCHI
WSR/XSR M0..3	⇒	ESYNC	⇒	RSR/XSR M0..3
WSR/XSR M0..3	⇒	ESYNC	⇒	MAC16 Option instructions

Table 9–251. T1050 Additional SYNC Requirements

		Section 5.3.4 on page 215		
WSR/XSR SAR	⇒	ESYNC	⇒	RSR/XSR SAR
WSR/XSR SAR	⇒	ESYNC	⇒	SLL/SRL/SRA/SRC
WSR/XSR BR	⇒	ESYNC	⇒	RSR/XSR BR
WSR/XSR BR	⇒	ESYNC	⇒	Listed instruction use of BR
Instruction setting of BR	⇒	ESYNC	⇒	RSR/XSR BR
WSR/XSR LITBASE	⇒	ESYNC	⇒	RSR/XSR LITBASE
WSR/XSR SCOMPARE1	⇒	ESYNC	⇒	RSR/XSR SCOMPARE1
		Section 5.3.5 on page 216		
WSR/XSR PS	⇒	ESYNC	⇒	RSR/XSR PS
WSR/XSR PS	⇒	RSYNC	⇒	CALL4/8/12, CALLX4/8/12
WSR/XSR PS	⇒	RSYNC	⇒	RFI/RFDD/RFDO/RFE/RFWO/RFWU/RSIL/WAITI
WSR/XSR PS.INTLEVEL	⇒	RSYNC	⇒	RSIL
WSR/XSR PS.UM	⇒	ESYNC	⇒	RSIL
WSR/XSR PS.RING	⇒	RSYNC	⇒	Privileged instruction exception
WSR/XSR PS.OWB	⇒	RSYNC	⇒	RFWO/RFWU
WSR/XSR PS.OWB	⇒	RSYNC	⇒	RSIL
WSR/XSR PS.CALLINC	⇒	RSYNC	⇒	ENTRY/RSIL
WSR/XSR PS.WOE	⇒	RSYNC	⇒	RSIL
		Section 5.3.6 on page 221		
WSR/XSR WINDOWBASE	⇒	ESYNC	⇒	RSR/XSR WINDOWBASE
WSR/XSR WINDOWSTART	⇒	ESYNC	⇒	RSR/XSR WINDOWSTART
		Section 5.3.7 on page 221		
WSR/XSR PTEVADDR	⇒	ESYNC	⇒	RSR/XSR PTEVADDR
WSR/XSR EXCVADDR	⇒	ESYNC	⇒	RSR/XSR PTEVADDR
WSR/XSR RASID	⇒	ESYNC	⇒	RSR/XSR RASID
WSR/XSR ITLBCFG	⇒	ESYNC	⇒	RSR/XSR ITLBCFG
WSR/XSR DTLBCFG	⇒	ESYNC	⇒	RSR/XSR DTLBCFG
		Section 5.3.8 on page 223		
WSR/XSR EXCCAUSE	⇒	ESYNC	⇒	RSR/XSR EXCCAUSE
WSR/XSR EXCVADDR	⇒	ESYNC	⇒	RSR/XSR EXCVADDR
WSR/XSR EXCVADDR	⇒	ESYNC	⇒	RSR/XSR PTEVADDR

Table 9–251. T1050 Additional SYNC Requirements

Section 5.3.9 on page 226				
WSR/XSR EPC1	⇒	ESYNC	⇒	RSR/XSR EPC1
WSR/XSR EPC1	⇒	ESYNC	⇒	RFE/RFWO/RFWU
WSR/XSR EPC2..7	⇒	ESYNC	⇒	RSR/XSR EPC2..7
WSR/XSR EPC2..7	⇒	ESYNC	⇒	RFI 2..7 (to the level of the EPC changed)
WSR/XSR DEPC	⇒	ESYNC	⇒	RSR/XSR DEPC
WSR/XSR DEPC	⇒	ESYNC	⇒	RFDE
WSR/XSR MEPC	⇒	(none)	⇒	RSR/XSR MEPC
WSR/XSR MEPC	⇒	(none)	⇒	RFME
WSR/XSR EPS2..7	⇒	ESYNC	⇒	RSR/XSR EPS2..7
WSR/XSR EPS2..7	⇒	RSYNC	⇒	RFI 2..7 (to the level of the EPS changed)
WSR/XSR EXCSAVE1	⇒	ESYNC	⇒	RSR/XSR EXCSAVE1
WSR/XSR EXCSAVE2..7	⇒	ESYNC	⇒	RSR/XSR EXCSAVE2..7 (to the same register)
WSR/XSR MESAVE	⇒	(none)	⇒	RSR/XSR MESAVE
Section 5.3.11 on page 231				
WSR/XSR ICOUNTLEVEL	⇒	ESYNC	⇒	RSR/XSR ICOUNTLEVEL
WSR/XSR CCOMPARE0..2	⇒	ESYNC	⇒	RSR/XSR CCOMPARE0..2
Section 5.3.12 on page 233				
WSR/XSR IBREAKENABLE	⇒	ESYNC	⇒	RSR/XSR IBREAKENABLE
WSR/XSR IBREAKA0..1	⇒	ESYNC	⇒	RSR/XSR IBREAKA0..1
WSR/XSR DBREAKC0..1	⇒	ESYNC	⇒	RSR/XSR DBREAKC0..1
WSR/XSR DBREAKA0..1	⇒	ESYNC	⇒	RSR/XSR DBREAKA0..1
Section 5.3.13 on page 235				
WSR/XSR MISC0..3	⇒	ESYNC	⇒	RSR/XSR MISC0..3
WSR/XSR CPENABLE	⇒	ESYNC	⇒	RSR/XSR CPENABLE
WSR/XSR CPENABLE	⇒	RSYNC	⇒	Any coprocessor instruction if its enable bit was changed

Index