ADSP-BF70x Blackfin+ Processor Programming Reference

Preliminary Revision 0.2, May 2014

Part Number
82-100123-01

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

Copyright Information

© 2014 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, Blackfin, EZ-KIT Lite, SHARC, TigerSHARC, VisualDSP++, and CrossCore Embedded Studio are registered trademarks of Analog Devices, Inc.

Blackfin+ is a trademark of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

Contents

Preface

Introduction

Computational Units

Operating Modes and States

Program Sequencer

Core Timer (TMR)

Address Arithmetic Unit

Memory

Instruction Reference Pages

Debug

Numeric Formats

Index

Preface

Thank you for purchasing and developing systems using an ADSP-BF70x Blackfin+ processor from Analog Devices, Inc.

Purpose of This Manual

The *ADSP-BF70x Blackfin+ Processor Programming Reference* provides core architecture and programming information about the ADSP-BF70x processors. This programming reference provides the main architectural information about the core of these processors. The architectural descriptions cover computational units, the control units, the address arithmetic unit, and control unit, including all features and processes that they support. For hardware (system design) information, see the *Blackfin+ Processor Hardware Reference*. For timing, electrical, and package specifications, see the *ADSP-BF70x Blackfin+ Processor Data Sheet*.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. The manual assumes the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts, such as programming reference books and data sheets, that describe their target architecture.

Manual Contents

This manual consists of the following chapters:

- **Introduction** - Provides a general description of the processor core architecture, memory architecture, instruction syntax, and notation conventions.
- **Computational Units** - Describes the arithmetic/logic units (ALUs), multiplier/accumulator units (MACs), shifter, and the set of video ALUs. The chapter also discusses data formats, data types, and register files.
- **Operating Modes and States** - Describes the operating modes of the processor. The chapter also describes Idle state and Reset state.

- **Program Sequencer** - Describes the operation of the program sequencer, which controls program flow by providing the address of the next instruction to be executed. The chapter also discusses loops, subroutines, jumps, interrupts, and exceptions.
- **Address Arithmetic Unit** - Describes the Address Arithmetic Unit (AAU), including Data Address Generators (DAGs), addressing modes, how to modify DAG and Pointer registers, memory address alignment, and DAG instructions.
- **Memory** - Describes L1 memories. In particular, details their memory architecture, memory model, memory transaction model, and memory-mapped registers (MMRs). Discusses the instruction, data, and scratchpad memory, which are part of the Blackfin processor core.
- Instruction Set Reference - Describes each instruction with a reference page (providing descriptions of assembly language instructions and describing their execution.); the reference contains the following instruction description sections:
  - Arithmetic Instructions
  - Sequencer Instructions
  - Memory or Pointer Instructions
  - Specialized Compute Instructions
  - Arithmetic Status Register
  - Instruction Page Tables (including instruction opcode information)
  - Issuing Parallel Instructions
- **Debug** - Provides a description of the processor's debug functionality, which is used for software debugging. This functionality also complements some services often found in an operating system (OS) kernel.
- **Numeric Formats** - Describes various aspects of the 16-bit data format. The chapter also describes how to implement a block floating-point format in software.

What's New in This Manual

This revision (0.1) is a preliminary revision of the *ADSP-BF70x Blackfin+ Processor Programming Reference*. This initial revision does not include complete content for all chapters.

Technical or Customer Support

You can reach customer and technical support for processors from Analog Devices in the following ways:

- Post your questions in the processors and DSP support community at **EngineerZone**:

  http://ez.analog.com/community/dsp

- Submit your questions to technical support at **Connect with ADI Specialists**:

  http://www.analog.com/support

- E-mail your questions about software/hardware development tools to:

  processor.tools.support@analog.com

- E-mail your questions about processors and DSPs to:

  processor.support@analog.com (world wide support)

  processor.china@analog.com (China support)

- Phone questions to *1-800-ANALOGD* (USA only)

- Contact your Analog Devices sales office or authorized distributor. Locate one at:

  http://www.analog.com/adi-sales

- Send questions by mail to:

  Analog Devices, Inc.
  Three Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106 USA

Supported Processors

The following is the list of Analog Devices, Inc. processors supported in CrossCore Embedded Studio® development tools.

Blackfin+® (ADSP-BF7xx) Processors

The name *Blackfin+* refers to a family of 16-bit, embedded processors. CrossCore Embedded Studio currently supports the following Blackfin+ processors: ADSP-BF702, ADSP-BF703, ADSP-BF704, ADSP-BF705, ADSP-BF706, and ADSP-BF707.

Blackfin® (ADSP-BF6xx/BF5xx) Processors

The name *Blackfin* refers to a family of 16-bit, embedded processors. CrossCore Embedded Studio currently supports the following Blackfin families ADSP-BF50x, ADSP-BF51x, ADSP-BF52x, ADSP-BF53x, ADSP-BF54x, ADSP-BF59x, ADSP-BF561, and ADSP-BF60x processors.

SHARC® (ADSP-21xxx) Processors

The name *SHARC* refers to a family of high-performance, 32-bit, floating-point processors that can be used in speech, sound, graphics, and imaging applications. CrossCore Embedded Studio currently supports the following SHARC families: ADSP-2106x, ADSP-2116x, ADSP-2126x, ADSP-2136x, and ADSP-214xx.

The following is the list of Analog Devices, Inc. processors supported in IAR Embedded WorkBench® development tools. For information about the IAR Embedded WorkBench product and software download, go to http://www.iar.com/en/Products/IAR-Embedded-Workbench

ADSP-CM40x Mixed-Signal Control Processors

The ADSP-CM40x processor is based on the ARM® Cortex®-M4 core and is designed for motor control and industrial applications. This product family includes the following mixed-signal control processors: ADSP-CM402F, ADSP-CM403F, ADSP-CM407F, and ADSP-CM408F.

Product Information

Product information can be obtained from the Analog Devices Web site and CrossCore Embedded Studio online Help system.

Analog Devices Web Site

The Analog Devices Web site, http://www.analog.com, provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

To access a complete technical library for each processor family, go to: http://www.analog.com/processors/technical_library The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information about products you are interested in. You can choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Visit MyAnalog.com to sign up. If you are a registered user, just log on. Your user name is your e-mail address.

EngineerZone

EngineerZone is a technical support forum from Analog Devices. It allows you direct access to ADI technical support engineers. You can search FAQs and technical information to get quick answers to your embedded processing and DSP design questions.

Use EngineerZone to connect with other DSP developers who face similar design challenges. You can also use this open forum to share knowledge and collaborate with the ADI support team and your peers. Visit http://ez.analog.com to sign up.

Notation Conventions

Text conventions used in this manual are identified and described as follows. Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the CrossCore Embedded Studio IDE's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `Letter Gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
NOTE: This is an note paragraph.	**Note:** For correct operation, ... A **Note** provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
CAUTION: This is a caution paragraph.	**Caution:** Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
DANGER: This is a danger paragraph.	**Danger:** Injury to device users may result if ... A **Danger** identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for the devices users. In the online version of this book, the word **Danger** appears instead of this symbol.

Register Documentation Conventions

Register diagrams use the following conventions:

- The descriptive name of the register appears at the top with the short form of the name.
- If a bit has a short name, the short name appears first in the bit description, followed by the long name.
- The reset value appears in binary in the individual bits and in hexadecimal to the left of the register.

- Bits marked **X** have an unknown reset value. Consequently, the reset value of registers that contain such bits is undefined or dependent on pin values at reset.
- Shaded bits are reserved.

NOTE: To ensure upward compatibility with future implementations, write back the value that is read for reserved bits in a register, unless otherwise specified.

Register description tables use the following conventions:

- Each bit's or bit field's access type appears beneath the bit number in the table in the form (read-access/write-access). The access types include:
  - R = read, RC = read clear, RS = read set, R0 = read zero, R1 = read one, Rx = read undefined
  - W = write, NW = no write, W1C = write one to clear, W1S = write one to set, W0C = write zero to clear, W0S = write zero to set, WS = write to set, WC = write to clear, W1A = write one action
- Many bit and bit field descriptions include enumerations, identifying bit values and related functionality. Unless otherwise indicated (with a prefix), these enumerations are decimal values.

1 Introduction

This *Blackfin Processor Programming Reference* provides details on the assembly language instructions used by Blackfin Processors. The Blackfin 2 Architecture extends the Micro Signal Architecture (MSA) core developed jointly by Analog Devices, Inc. and Intel Corporation. This manual is applicable to all ADSP-BF7xx processor derivatives. All devices provide an identical core architecture and instruction set. Additional architectural features are only supported by some devices. These are identified in the manual as optional features. A read only memory mapped register, FEATURE0, enables software to query, at runtime, the optional features implemented in a particular derivative. Some details of the implementation may vary between derivatives. Generally this is not visible to software, but system and test software may depend very specific aspects of the memory microarchitecture. Differences and commonalities at a global level are discussed in the Memory chapter. For a full description of the system architecture beyond the Blackfin core, refer to the specific hardware reference for your derivative. This section points out some of the conventions used in this document.

The Blackfin processor combines a dual MAC signal processing engine, an orthogonal RISC-like microprocessor instruction set, flexible Single Instruction, Multiple Data (SIMD) capabilities, and multimedia features into a single instruction set architecture.

Core Architecture

The Blackfin processor core contains two 16-bit multipliers, one 32-bit multiplier, two 40-bit accumulators, one 72-bit accumulator, two 40-bit arithmetic logic units (ALUs), four 8-bit video ALUs, and a 40-bit shifter, shown in the **Processor Core Architecture** figure. The Blackfin processors process 8-, 16-, or 32-bit data from the register file.

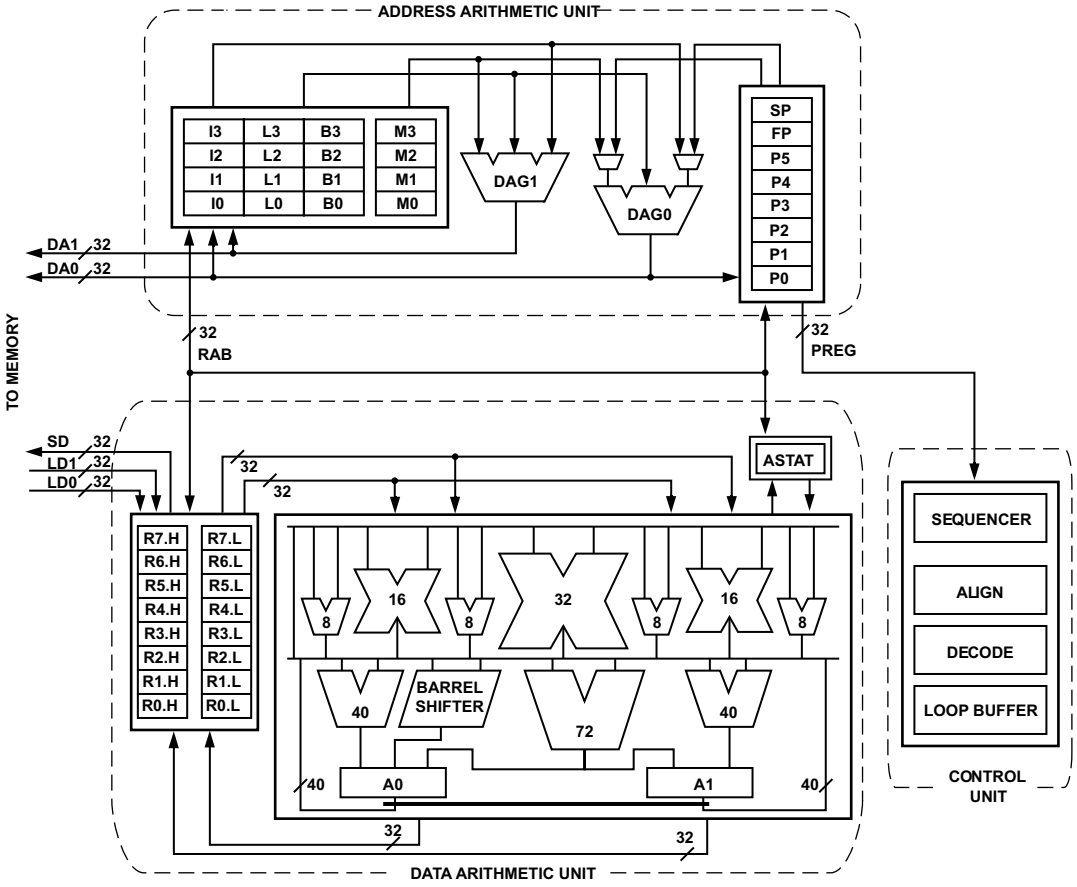


Figure 1-1: Processor Core Architecture

The compute register file contains eight 32-bit registers. When performing compute operations on 16-bit operand data, the register file operates as 16 independent 16-bit registers. All operands for compute operations come from the multiported register file and instruction constant fields.

Each 16-bit MAC can perform a 16- by 16-bit multiply per cycle, with accumulation to a 40-bit result. The 32-bit MAC can perform a 32- by 32- multiply, with accumulation to 72-bits, or a 16-bit complex multiplication. Signed and unsigned formats, rounding, and saturation are supported.

The ALUs perform a traditional set of arithmetic and logical operations on 16-bit or 32-bit data. Many special instructions are included to accelerate various signal processing tasks. These include bit operations such as field extract and population count, divide primitives, saturation and rounding, and sign/exponent detection. The set of video instructions include byte alignment and packing operations, 16-bit and 8-bit adds with clipping, 8-bit average operations, and 8-bit subtract/absolute value/accumulate (SAA) operations. Also provided are the compare/select and vector search instructions. For some instructions, two 16-bit ALU operations can be performed simultaneously on register pairs (a 16-bit high half and 16-bit low half of a compute register). By also using the second ALU, quad 16-bit operations are possible.

The 40-bit shifter can deposit data and perform shifting, rotating, normalization, and extraction operations.

A program sequencer controls the instruction execution flow, including instruction alignment and decoding. For program flow control, the sequencer supports PC-relative and indirect conditional jumps (with static branch prediction) and subroutine calls. Hardware is provided to support zero-overhead looping. The architecture is fully interlocked, meaning there are no visible pipeline effects when executing instructions with data dependencies.

The address arithmetic unit provides two addresses for simultaneous dual fetches from memory. It contains a multiported register file consisting of four sets of 32-bit Index, Modify, Length, and Base registers (for circular buffering) and eight additional 32-bit pointer registers (for C-style indexed stack manipulation).

Blackfin processors support a modified Harvard architecture in combination with a hierarchical memory structure. Level 1 (L1) memories typically operate at the full processor speed with little or no latency. At the L1 level, the instruction memory holds instructions only. The two data memories hold data, and a dedicated scratchpad data memory stores stack and local variable information.

In addition, multiple L1 memory blocks are provided, which may be configured as a mix of SRAM and cache. The Memory Management Unit (MMU) provides memory protection for individual tasks that may be operating on the core.

The architecture provides three modes of operation: User, Supervisor, and Emulation. User mode has restricted access to a subset of system resources, thus providing a protected software environment. Supervisor and Emulation modes have unrestricted access to the system and core resources.

The Blackfin processor instruction set is optimized so that 16-bit opcodes represent the most frequently used instructions. Complex DSP instructions are encoded into 32-bit opcodes as multifunction instructions, and some instructions with very large immediate values are encoded into 64-bit opcodes. Blackfin products support a limited multi-issue capability, where a 32-bit instruction can be issued in parallel with two 16-bit instructions. This allows the programmer to use many of the core resources in a single instruction cycle.

The Blackfin processor assembly language uses an algebraic syntax. The architecture is optimized for use with the C compiler.

Memory Architecture

The Blackfin processor architecture structures memory as a single, unified 4G byte address space using 32-bit addresses, regardless of the specific Blackfin product. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this common address space. The memory portions of this address space are arranged in a hierarchical structure to provide a good cost/performance balance of some very fast, low latency on-chip memory as cache or SRAM, and larger, lower cost and lower performance off-chip memory systems.

The memory DMA controller provides high bandwidth data movement capability. It can perform block transfers of code or data between the internal memory and the external memory spaces.

Internal Memory

The L1 memory system is the primary highest performance memory available to the core. At a minimum, each Blackfin processors has two blocks of on-chip memory that provide high bandwidth access to the core:

- L1 instruction memory, consisting of SRAM and, optionally, an instruction cache. This memory is accessed at full processor speed.
- L1 data memory, consisting of SRAM and/or a data cache. This memory block is also accessed at full processor speed.

Some Blackfin systems contain additional on-chip memories. On-chip Level 2 (L2) memory forms an on-chip memory hierarchy with L1 memory and provides much more capacity than L1 memory, but the latency is higher. The on-chip L2 memory may be SRAM or cache. On-chip L2 memory is capable of storing both instructions and data and may be accessible by more than one core in a multi-core system.

External Memory

External (off-chip) memory is accessed via on-chip memory peripherals such as DDR controllers.

I/O Memory Space

Blackfin processors do not define a separate I/O space. All resources are mapped through the flat 32-bit address space. Control registers for on-chip I/O devices are mapped into memory-mapped registers (MMRs) at addresses in a reserved part of the 4G byte address space. These are separated into two smaller blocks: one contains the control MMRs for all core functions and the other contains the registers needed for setup and control of the on-chip peripherals outside of the core. The MMRs are accessible only in Supervisor mode.

Event Handling

The event controller on the Blackfin processor handles all asynchronous and synchronous events to the processor. The processor event handling supports both nesting and prioritization. Nesting allows multiple event service routines to be active simultaneously. Prioritization ensures that servicing a higher priority

event takes precedence over servicing a lower priority event. The controller provides support for five different types of events:

- Emulation - Causes the processor to enter Emulation mode, allowing command and control of the processor via the JTAG interface.
- Reset - Resets the processor.
- Nonmaskable Interrupt (NMI) - The software watchdog timer or the NMI input signal to the processor generates this event. The NMI event is frequently used as a power-down indicator to initiate an orderly shutdown of the system.
- Exceptions - Synchronous to program flow. That is, the exception is taken before the instruction is allowed to complete. Conditions such as data alignment violations and undefined instructions cause exceptions.
- Interrupts - Asynchronous to program flow. These are caused by input pins, timers, and other peripherals.

Each event has an associated register to hold the return address and an associated return-from-event instruction. When an event is triggered, the state of the processor is saved on the supervisor stack.

The processor event controller consists of two stages: the Core Event Controller (CEC) and the System Event Controller (SEC). The CEC works with the SEC to prioritize and control all system events. Conceptually, interrupts from the peripherals arrive at the SEC and are routed directly into a general-purpose interrupt of the CEC.

Syntax Conventions

The Blackfin processor instruction set supports several syntactic conventions that appear throughout this document. Those conventions relate to case sensitivity, free format, instruction delimiting, and comments.

Case Sensitivity

The instruction syntax is case insensitive. Upper and lower case letters can be used and intermixed arbitrarily.

The assembler treats register names and instruction keywords in a case-insensitive manner. User identifiers are case sensitive. Thus, `R3.l`, `R3.L`, `r3.l`, `r3.L` are all valid, equivalent input to the assembler.

This manual shows register names and instruction keywords in examples using lower case. Otherwise, in explanations and descriptions, this manual uses upper case to help the register names and keywords stand out among text.

Free Format

Assembler input is free format, and may appear anywhere on the line. One instruction may extend across multiple lines, or more than one instruction may appear on the same line. White space (space, tab, comments, or newline) may appear anywhere between tokens. A token must not have embedded spaces. Tokens include numbers, register names, keywords, user identifiers, and also some multicharacter special symbols like "+=", "`/*`", or "`||`".

Instruction Delimiting

A semicolon must terminate every instruction. Several instructions can be placed together on a single line at the programmer's discretion, provided each instruction ends with a semicolon.

Each complete instruction must end with a semicolon. Sometimes, a complete instruction will consist of more than one operation. There are two cases where this occurs.

- Two general operations are combined. Normally a comma separates the different parts, as in

  ```
  a0 = r3.h * r2.l , a1 = r3.l * r2.h ;
  ```

- A general instruction is combined with one or two memory references for joint issue. The latter portions are set off by a "`||`" token. For example,

  ```
  a0 = r3.h * r2.l || r1 = [p3++] || r4 = [i2++] ;
  ```

Comments

The assembler supports various kinds of comments, including the following.

- End of line: A double forward slash token ("`//`") indicates the beginning of a comment that concludes at the next newline character.
- General comment: A general comment begins with the token "`/*`" and ends with "`*/`". It may contain any characters and extend over multiple lines.

Comments are not recursive; if the assembler sees a "`/*`" within a general comment, it issues an assembler warning. A comment functions as white space.

Notation Conventions

This manual and the assembler use the following conventions.

- Register names are alphabetical, followed by a number in cases where there are more than one register in a logical group. Thus, examples include `ASTAT`, `FP`, `R3`, and `M2`.
- Register names are reserved and may not be used as program identifiers.

- Some operations (such as the *Move Register* instruction) require a register pair. Register pairs are always Data Registers or Accumulators and are denoted using a colon, for example, `R3:2`. The larger number must be written first. Note that the hardware supports only odd-even pairs, for example, `R7:6`, `R5:4`, `R3:2`, `R1:0 and A1:0`.
- Some instructions (such as the *--SP (Push Multiple)* instruction) require a group of adjacent registers. Adjacent registers are denoted in syntax by the range enclosed in parentheses and separated by a colon, for example, `(R7:3)`. Again, the larger number appears first.
- Portions of a particular register may be individually specified. This is written in syntax with a dot (".") following the register name, then a letter denoting the desired portion. For 32-bit registers, ".`H`" denotes the most-significant ("High") portion, ".`L`" denotes the least-significant portion. The subdivisions of the 40-bit registers are described later.

Register names are reserved and may not be used as program identifiers.

This manual uses the following conventions.

- When there is a choice of any one register within a register group, this manual shows the register set using an en-dash ("-"). For example, "`R7-0`" in text means that any one of the eight data registers (`R7`, `R6`, `R5`, `R4`, `R3`, `R2`, `R1`, or `R0`) can be used in syntax.
- Immediate values are designated as "`imm`" with the following modifiers.
  - "`imm`" indicates a signed value; for example, *`imm7`*.
  - T he "`u`" prefix indicates an unsigned value; for example, *`uimm4`*.
  - The decimal number indicates how many bits the value can include; for example, *`imm5`* is a 5-bit value.
  - Any alignment requirements are designated by an optional "`m`" suffix followed by a number; for example, *`uimm16m2`* is an unsigned, 16-bit integer that must be an even number, and *`imm7m4`* is a signed, 7-bit integer that must be a multiple of 4.
  - PC-relative, signed values are designated as "*`pcrel`*" with the following modifiers:
    - the decimal number indicates how many bits the value can include; for example, *`pcrel5`* is a 5-bit value.
    - any alignment requirements are designated by an optional "m" suffix followed by a number; for example, *`pcrel13m2`* is a 13-bit integer that must be an even number.
  - Loop PC-relative, signed values are designated as "*`lppcrel`*" with the following modifiers:
    - the decimal number indicates how many bits the value can include; for example, *`lppcrel5`* is a 5-bit value.
    - any alignment requirements are designated by an optional "m" suffix followed by a number; for example, *`lppcrel11m2`* is an 11-bit integer that must be an even number.

Glossary

The following terms appear throughout this document. Without trying to explain the Blackfin processor, here are the terms used with their definitions.

Register Names

The architecture includes the registers shown in the **Registers** table.

Table 1-1: Registers

Register	Description
Accumulators	The set of 40-bit registers A1 and A0 that normally contain data that is being manipulated. Each Accumulator can be accessed in five ways: as one 40-bit register, as one 32-bit register (designated as A1.W or A0.W), as two 16-bit registers similar to Data Registers (designated as A1.H, A1.L, A0.H, or A0.L) and as one 8-bit register (designated A1.X or A0.X) for the bits that extend beyond bit 31. The accumulators may be used as a pair (designated A1:0) to hold the 40-bit real and imaginary parts of a complex fixed-point number, or a 72-bit real fixed-point number.
Data Registers	The set of 32-bit registers (R0, R1, R2, R3, R4, R5, R6, and R7) that normally contain data for manipulation. Abbreviated data register or Dreg. Data registers can be accessed as 32-bit registers, or optionally as two independent 16-bit registers. The least significant 16 bits of each register is called the "low" half and is designated with ".L" following the register name. The most significant 16 bit is called the "high" half and is designated with ".H" following the name. Example: R7.L, r2.h, r4.L, R0.h.
Pointer Registers	The set of 32-bit registers (P0, P1, P2, P3, P4, P5, including SP and FP) that normally contain byte addresses of data structures. Accessed only as a 32-bit register. Abbreviated pointer register or Preg. Example: p2, p5, fp, sp.
Stack Pointer	SP; contains the 32-bit address of the last occupied byte location in the stack. The stack grows by decrementing the Stack Pointer. A subset of the Pointer Registers.
Frame Pointer	FP; contains the 32-bit address of the previous Frame Pointer in the stack, located at the top of a frame. A subset of the Pointer Registers.
Loop Top	LT0 and LT1; contains 32-bit address of the top of a zero overhead loop.
Loop Count	LC0 and LC1; contains 32-bit counter of the zero overhead loop executions.
Loop Bottom	LB0 and LB1; contains 32-bit address of the bottom of a zero overhead loop.
Index Register	The set of 32-bit registers I0, I1, I2, I3 that normally contain byte addresses of data structures. Abbreviated index register or Ireg.
Modify Registers	The set of 32-bit registers M0, M1, M2, M3 that normally contain offset values that are added or subtracted to one of the Index Registers. Abbreviated as Mreg.
Length Registers	The set of 32-bit registers L0, L1, L2, L3 that normally contain the length (in bytes) of the circular buffer. Abbreviated as Lreg. Clear Lreg to disable circular addressing for the corresponding Ireg. Example: Clear L3 to disable circular addressing for I3.
Base Registers	The set of 32-bit registers B0, B1, B2, B3 that normally contain the base address (in bytes) of the circular buffer. Abbreviated as Breg.

Functional Units

The architecture includes the three processor sections shown in the **Units** table.

Table 1-2: Units

Processor	Description
Data Address Generator (DAG)	Calculates the effective address for indirect and indexed memory accesses. Consists of two sections-DAG0 and DAG1.
Multiply and Accumulate Unit (MAC)	Performs the arithmetic functions on data. Consists of three sections: MAC0 and MAC1, each associated with an Accumulator (A0 and A1, respectively), and MAC10 associated with the accumulator pair A1:0.
Arithmetic Logical Unit (ALU)	Performs arithmetic computations and binary shifts on data. Operates on the Data Registers and Accumulators. Consists of three units (ALU0, ALU1 and ALU10), each associated with an Accumulator (A0, A1 and A1:0, respectively). Each ALU operates in conjunction with a Multiply and Accumulate Unit.

Arithmetic Status Bits

The Blackfin architecture includes 12 arithmetic status bits (status bits) that indicate specific results of a prior operation. These bits reside in the Arithmetic Status (ASTAT) Register. A summary of the status bits appears in the **ASTAT Bits** table. All status bits are active high. Instructions regarding pointer registers, index registers, length registers, modify registers, or base registers do not affect status bits.

Table 1-3: ASTAT Bits

Bit	Description
AC0	Carry (ALU0)
AC0_COPY	Carry (ALU0), copy
AC1	Carry (ALU1)
AN	Negative
AQ	Quotient
AV0	Accumulator 0 Overflow
AVS0	Accumulator 0 Sticky Overflow Set when AV0 is set, but remains set until explicitly cleared by user code.
AV1	Accumulator 1 Overflow
AVS1	Accumulator 1 Sticky Overflow Set when AV1 is set, but remains set until explicitly cleared by user code.
AZ	Zero

Table 1-3: ASTAT Bits (Continued)

Bit	Description
CC	Control Code bit Multipurpose bit set, cleared and tested by specific instructions.
V	Overflow for Data Register results
V_COPY	Overflow for Data Register results. copy
VS	Sticky Overflow for Data Register results Set when V is set, but remains set until explicitly cleared by user code.

Fractional Convention

Fractional numbers include subinteger components less than 1. Whereas decimal fractions appear to the right of a decimal point, binary fractions appear to the right of a binal point.

In DSP instructions that assume placement of a binal point, for example in computing sign bits for normalization or for alignment purposes, the binal point convention depends on the size of the register being used as shown in the **Fractional Notation** table and the **Conventional Placement of Binal Point** figure.

Table 1-4: Fractional Notation

Registers Size	Format	Notation	Sign Bit	Extension Bits	Fractional Bits
80-bit accumulator pair	Signed Fractional	9.63	1	8	63
	Unsigned Fractional	8.64	0	8	64
64-bit register pair	Signed Fractional	1.63	1	0	63
	Unsigned Fractional	0.64	0	0	64
40-bit registers	Signed Fractional	9.31	1	8	31
	Unsigned Fractional	8.32	0	8	32
32-bit registers	Signed Fractional	1.31	1	0	31
	Unsigned Fractional	0.32	0	0	32
16-bit registers	Signed Fractional	1.15	1	0	15
	Unsigned Fractional	0.16	0	0	16

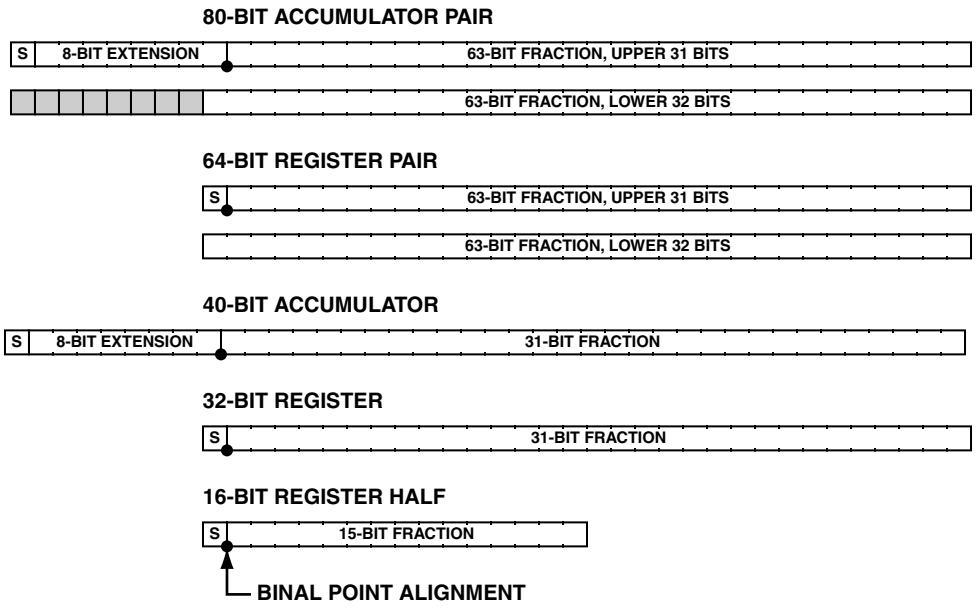


Figure 1-2: Conventional Placement of Binal Point

Saturation

When the result of an arithmetic operation exceeds the range of the destination register, important information can be lost.

Saturation is a technique used to contain the quantity within the values that the destination register can represent. When a value is computed that exceeds the capacity of the destination register, then the value written to the register is the largest value that the register can hold with the same sign as the original.

- If an operation would otherwise cause a positive value to overflow and become negative, instead, saturation limits the result to the maximum positive value for the size register being used.
- Conversely, if an operation would otherwise cause a negative value to overflow and become positive, saturation limits the result to the maximum negative value for the register size.

The maximum positive value in a 16-bit register is 0x7FFF. The maximum negative value is 0x8000. For a signed two's-complement 1.15 fractional notation, the allowable range is -1 through ($1-2^{-15}$).

The maximum positive value in a 32-bit register is 0x7FFF FFFF. The maximum negative value is 0x8000 0000. For a signed two's-complement fractional data in 1.31 format, the range of values that the register can hold are -1 through ($1-2^{-31}$).

The maximum positive value in a 40-bit register is 0x7F FFFF FFFF. The maximum negative value is 0x80 0000 0000. For a signed two's-complement 9.31 fractional notation, the range of values that can be represented is -256 through ($256-2^{-31}$).

The maximum positive value in a 64-bit register pair is 0x7FFF FFFF FFFF FFFF. The maximum negative value is 0x8000 0000 0000 0000. For a signed two's-complement fractional data in 1.63 format, the range of values that the register can hold are -1 through ($1-2^{-63}$).

A real value held in the 80-bit accumulator pair, A1:0, only has 72-bit of useful data. So the maximum positive value in the 80-bit accumulator pair is 0x7F FFFF FFFF FFFF FFFF. The maximum negative value is 0x80 0000 0000 0000 0000. For a signed two's-complement 9.63 fractional notation, the range of values that can be represented is -256 through ($256-2^{-63}$).

For example, if a 16-bit register containing 0x1000 (decimal integer +4096) was shifted left 3 places without saturation, it would overflow to 0x8000 (decimal -32,768). With saturation, however, a left shift of 3 or more places would always produce the largest positive 16-bit number, 0x7FFF (decimal +32,767).

Another common example is copying the lower half of a 32-bit register into a 16-bit register. If the 32-bit register contains 0xFEED 0ACE and the lower half of this negative number is copied into a 16-bit register without saturation, the result is 0x0ACE, a positive number. But if saturation is enforced, the 16-bit result maintains its negative sign and becomes 0x8000.

The Blackfin architecture implements 40-bit saturation for all arithmetic operations that write a single Accumulator destination except as noted in the individual instruction descriptions when an optional 32-bit saturation mode can constrain a 40-bit Accumulator to the 32-bit register range. The Blackfin architecture performs 32-bit saturation for 32-bit register destinations only as noted in the instruction descriptions.

Overflow is the alternative to saturation. The number is allowed to simply exceed its bounds and lose its most significant bit(s); only the lowest (least-significant) portion of the number can be retained. Overflow can occur when a 40-bit value is written to a 32-bit destination, or a 72-bit value is written to a 64-bit or 32-bit location. If there was any useful information in the upper 8 bits of the 40-bit value, then information is lost in the process. Some processor instructions report overflow conditions in the arithmetic status bits, as noted in the instruction descriptions. The arithmetic status bits reside in the Arithmetic Status (ASTAT) Register.

Rounding and Truncating

Rounding is a means of reducing the precision of a number by removing a lower-order range of bits from that number's representation and possibly modifying the remaining portion of the number to more accurately represent its former value. For example, the original number will have N bits of precision, whereas the new number will have only M bits of precision (where N>M), so N-M bits of precision are removed from the number in the process of rounding.

The *round-to-nearest* method returns the closest number to the original. By convention, an original number lying exactly halfway between two numbers always rounds up to the larger of the two. For example, when rounding the 3-bit, two's-complement fraction 0.25 (binary 0.01) to the nearest 2-bit two's-complement fraction, this method returns 0.5 (binary 0.1). The original fraction lies exactly midway between 0.5 and 0.0 (binary 0.0), so this method rounds up. Because it always rounds up, this method is called *biased rounding*.

The *convergent rounding* method also returns the closest number to the original. However, in cases where the original number lies exactly halfway between two numbers, this method returns the nearest even number, the one containing an LSB of 0. So for the example above, the result would be 0.0, since that is the even numbered choice of 0.5 and 0.0. Since it rounds up and down based on the surrounding values, this method is called *unbiased rounding*.

Some instructions for this processor support biased and unbiased rounding. The `RND_MOD` bit in the Arithmetic Status (`ASTAT`) Register determines which mode is used.

Another common way to reduce the significant bits representing a number is to simply mask off the N-M lower bits. This process is known as *truncation* and results in a relatively large bias.

The **8-Bit Number Reduced to 4 Bits of Precision** figure shows other examples of rounding and truncation methods.

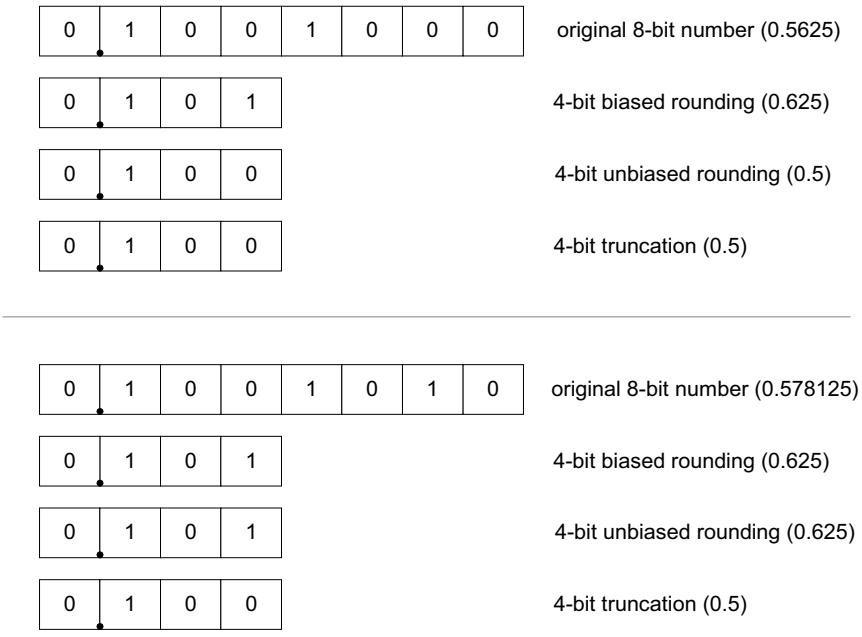


Figure 1-3: 8-Bit Number Reduced to 4 Bits of Precision

Automatic Circular Addressing

The Blackfin processor provides an optional circular (or "modulo") addressing feature that increments an Index Register (*`Ireg`*) through a predefined address range, then automatically resets the *`Ireg`* to repeat that range. This feature improves input/output loop performance by eliminating the need to manually reinitialize the address index pointer each time. Circular addressing is useful, for instance, when repetitively loading or storing a string of fixed-sized data blocks.

The circular buffer contents must meet the following conditions:

- The maximum length of a circular buffer (that is, the value held in any L register) must be an unsigned number with magnitude less than 2^{31}.
- The magnitude of the modifier should be less than the length of the circular buffer.
- The initial location of the pointer I should be within the circular buffer defined by the base B and length L.

If any of these conditions is not satisfied, then processor behavior is not specified.

There are two elements of automatic circular addressing:

- Indexed address instructions
- Four sets of circular addressing buffer registers composed of one each *Ireg*, *Breg*, and *Lreg* (i.e., I0/B0/L0, I1/B1/L1, I2/B2/L2, and I3/B3/L3)

To qualify for circular addressing, the indexed address instruction must explicitly modify an Index Register. Some indexed address instructions use a Modify Register (*Mreg*) to increment the *Ireg* value. In that case, any *Mreg* can be used to increment any *Ireg*. The *Ireg* used in the instruction specifies which of the four circular buffer sets to use.

The circular buffer registers define the length (*Lreg*) of the data block in bytes and the base (*Breg*) address to reinitialize the *Ireg*.

Some instructions modify an Index Register without using it for addressing; for example, the *Add Immediate* and *Modify–Decrement* instructions. Such instructions are still affected by circular addressing, if enabled.

Disable circular addressing for an *Ireg* by clearing the *Lreg* that corresponds to the *Ireg* used in the instruction. For example, clear L2 to disable circular addressing for register I2. Any nonzero value in an *Lreg* enables circular addressing for its corresponding buffer registers.

2 Computational Units

The processor's computational units perform numeric processing for DSP and general control algorithms. The seven computational units are two arithmetic/logic units (ALUs), three multiplier/accumulator (multiplier) units, a shifter, and a set of video ALUs. These units get data from registers in the Data Register File. Computational instructions for these units provide fixed-point operations, and each computational instruction can execute every cycle.

The computational units handle different types of operations. The ALUs perform arithmetic and logic operations. The multipliers perform multiplication and execute multiply/add and multiply/subtract operations. The shifter executes logical shifts and arithmetic shifts and performs bit packing and extraction. The video ALUs perform Single Instruction, Multiple Data (SIMD) logical operations on specific 8-bit data operands.

Data moving in and out of the computational units goes through the Data Register File, which consists of eight registers, each 32 bits wide. In operations requiring 16-bit operands, the registers are paired, providing sixteen possible 16-bit registers.

The processor's assembly language provides access to the Data Register File. The syntax lets programs move data to and from these registers and specify a computation's data format at the same time.

The **Processor Core Architecture** figure provides a graphical guide to the other topics in this chapter. An examination of each computational unit provides details about its operation and is followed by a summary of computational instructions. Studying the details of the computational units, register files, and data buses leads to a better understanding of proper data flow for computations. Next, details about the processor's advanced parallelism reveal how to take advantage of multifunction instructions.

The **Processor Core Architecture** figure shows the relationship between the Data Register File and the computational units-multipliers, ALUs, and shifter.

Single function multiplier, ALU, and shifter instructions have unrestricted access to the data registers in the Data Register File. Multifunction operations may have restrictions that are described in the section for that particular operation.

Two additional 40-bit registers, A0 and A1, provide accumulator results. These registers are dedicated to the ALUs and are used primarily for multiply-and-accumulate functions.

The traditional modes of arithmetic operations, such as fractional and integer, are specified directly in the instruction. Rounding modes are set from the ASTAT register, which also records status and conditions for the results of the computational operations.

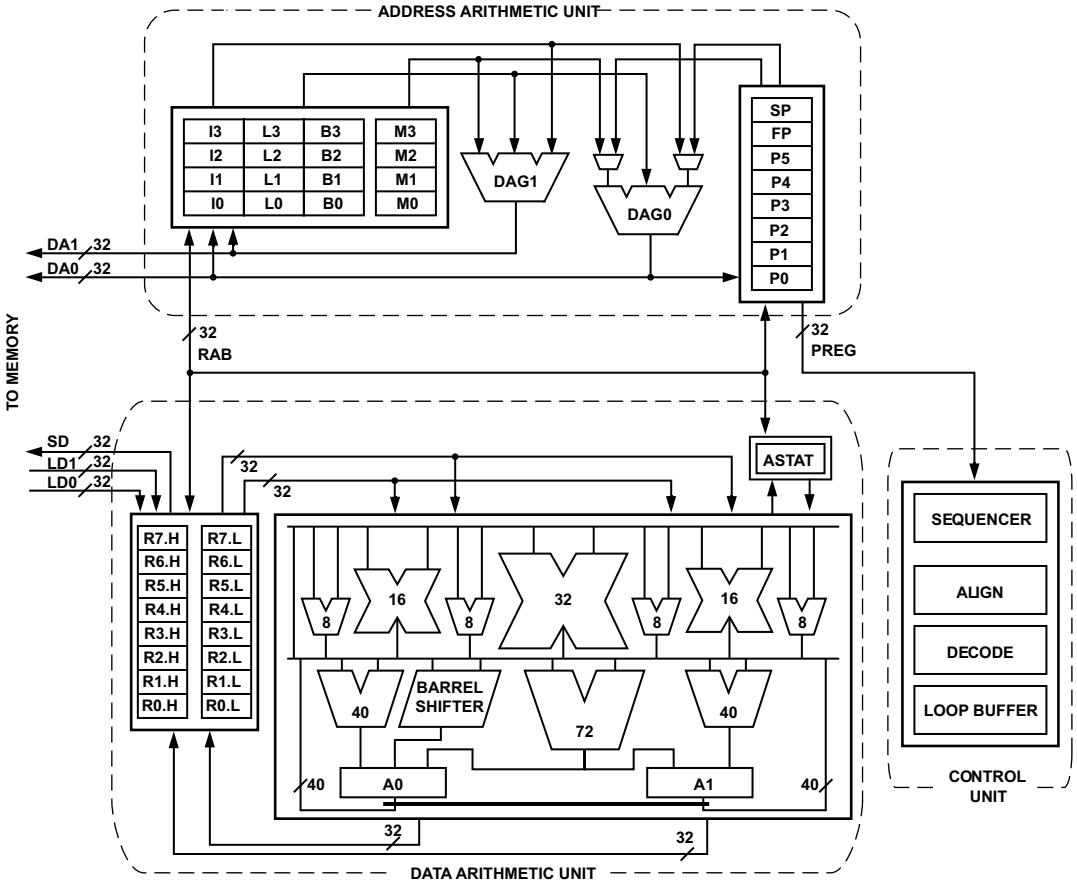


Figure 2-1: Processor Core Architecture

Using Data Formats

Blackfin processors are primarily 32-bit and 16-bit, fixed-point machines. Most operations assume a two's-complement number representation, while others assume unsigned numbers or simple binary strings. Other instructions support 32-bit complex fixed-point, 8-bit arithmetic and block floating point. For detailed information about each number format, see the numeric formats appendix.

In the Blackfin processor family arithmetic, signed numbers are always in two's-complement format. These processors do not use signed-magnitude, one's-complement, binary-coded decimal (BCD), or excess-n formats.

Binary String

The binary string format is the least complex binary notation; in it, 16 or 32 bits are treated as a bit pattern. Examples of computations using this format are the logical operations NOT, AND, OR, XOR. These ALU operations treat their operands as binary strings with no provision for sign bit or binary point placement.

Unsigned

Unsigned binary numbers may be thought of as positive and having nearly twice the magnitude of a signed number of the same length. The processor treats the least significant words of multiple precision numbers as unsigned numbers.

Signed Numbers: Two's-Complement

In Blackfin processor arithmetic, the word *signed* refers to two's-complement numbers. Most Blackfin processor family operations presume or support two's-complement arithmetic.

Fractional Representation: 1.15 and 1.31

Blackfin processor arithmetic is optimized for numerical values in a fractional binary format denoted by 1.15 ("one dot fifteen") and 1.31 ("one dot thirty one"). In the 1.15 format, 1 sign bit (the Most Significant Bit (MSB)) and 15 fractional bits represent values from -1 to 0.999969. In 1.31 format, 1 sign bit and 31 fractional bits represent values from -1 to 0.99999999953.

The **Bit Weighting for 1.15 Numbers and 1.31 Numbers** figure shows the bit weighting for fractional numbers. This figure also includes examples of 1.15 numbers and 1.31 numbers with their decimal equivalents.

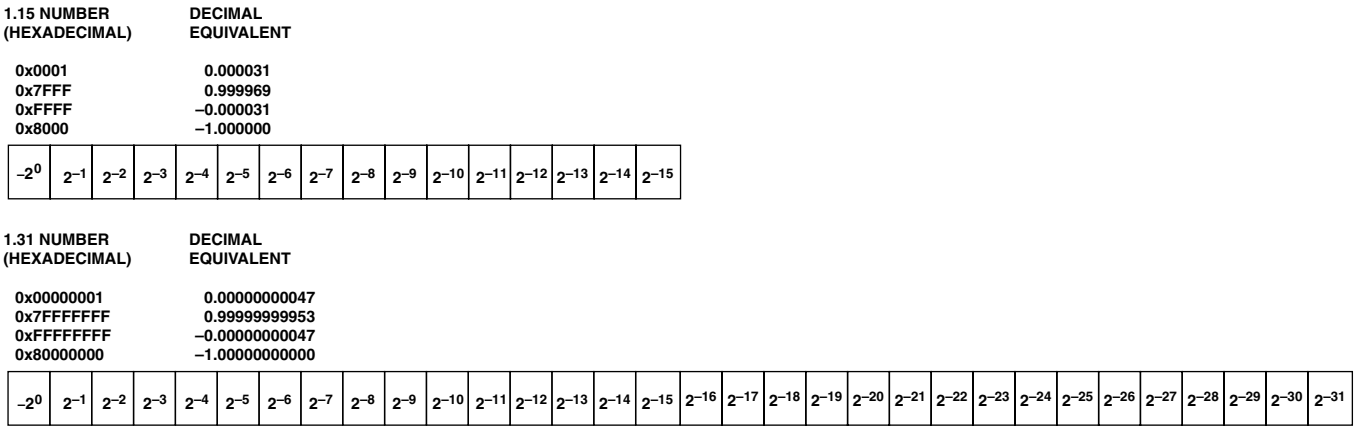


Figure 2-2: Bit Weighting for 1.15 Numbers and 1.31 Numbers

Complex Numbers

Complex numbers are represented as two 16-bit fixed point numbers. Either fractional in 1.15 format, or 16-bit signed integer. A complex operand is stored in a 32-bit Data register with the real part stored in the least significant bits and the imaginary part in the most significant bits.

Register Files

The processor's computational units have three definitive register groups-a Data Register File, a Pointer Register File, and set of Data Address Generation (DAG) registers.

- The Data Register File receives operands from the data buses for the computational units and stores computational results.
- The Pointer Register File has pointers for addressing operations.
- The DAG registers are dedicated registers that manage zero-overhead circular buffers for DSP operations.

The **AAU Register Files** figure provides more information on Pointer and DAG registers.

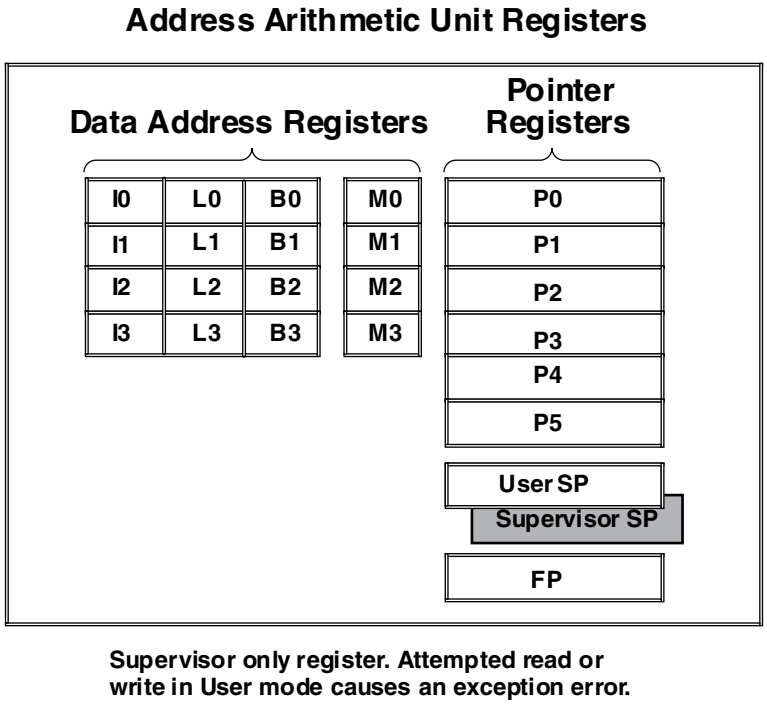


Figure 2-3: AAU Register Files

Data Register File

The Data Register File consists of eight registers, each 32 bits wide. Each register may be viewed as a pair of independent 16-bit registers. Each is denoted as the low half or high half. Thus the 32-bit register `R0` may be regarded as two independent register halves, `R0.L` and `R0.H`. For more information, see Data Register.

For example, these instructions represent a 32-bit and a 16-bit operation:

```
R2 = R1 + R2; /* 32-bit addition */
R2.L = R1.H * R0.L; /* 16-bit multiplication */
```

Three separate buses (two load, one store) connect the Register File to the L1 data memory, each bus being 32 bits wide. Transfers between the Data Register File and the data memory can move up to two 32-bit words of valid data in each cycle. Often, these represent four 16-bit words.

Accumulator Registers

In addition to the Data Register File, the processor has two dedicated, 40-bit accumulator registers, called `A0` and `A1`. Each can be referred to as its 16-bit low half (`An.L`) or high half (`An.H`) plus its 8-bit extension (`An.X`). Each can also be referred to as a 32-bit register (`An.W`) consisting of the lower 32 bits, or as a complete 40-bit result register (`An`). For more information, see Accumulator 0 Register.

These examples illustrate this convention:

```
A0 = A1; /* 40-bit move */
A1.W = R7; /* 32-bit move */
A0.H = R5.H; /* 16-bit move */
R6.H = A0.X; /* read 8-bit value and sign extend to 16 bits */
```

The accumulator registers may be use together to hold an 80-bit complex result or a 72-bit fixed point result. The combined accumulator register is called `A1:0`. A 72-bit fixed point result is held in the combined register with least significant bits in `A0.W`. middle bits in `A1.W` and most significant bits in `A1.X`.

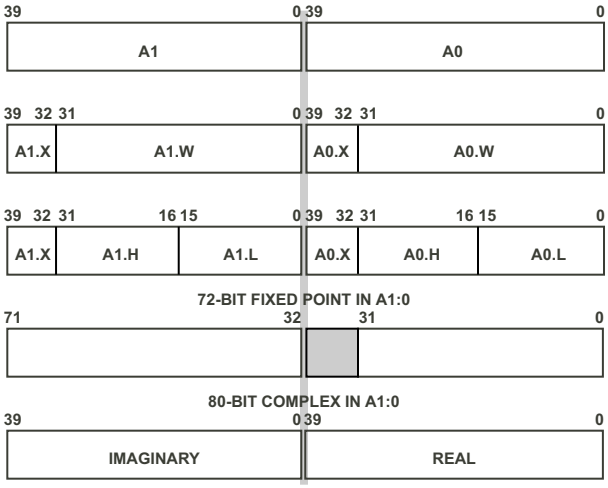


Figure 2-4: 40-Bit Accumulator Registers

Register File Instruction Summary

The **Register File Instructions and Status** table lists the register file instructions. In this table, note the meaning of these symbols:

- Allreg denotes: `R[7:0]`, `P[5:0]`, `SP`, `FP`, `I[3:0]`, `M[3:0]`, `B[3:0]`, `L[3:0]`, `A0.X`, `A0.W`, `A1.X`, `A1.W`, `ASTAT`, `RETS`, `RETI`, `RETX`, `RETN`, `RETE`, `LC[1:0]`, `LT[1:0]`, `LB[1:0]`, `USP`, `SEQSTAT`, `SYSCFG`, `EMUDAT`, `CYCLES`, and `CYCLES2`.
- Ax denotes either ALU Result register `A0` or `A1`.
- Dreg denotes any Data Register File register.
- Sysreg denotes the system registers: `ASTAT`, `SEQSTAT`, `SYSCFG`, `RETI`, `RETX`, `RETN`, `RETE`, or `RETS`, `LC[1:0]`, `LT[1:0]`, `LB[1:0]`, `EMUDAT`, `CYCLES`, and `CYCLES2`.
- Preg denotes any Pointer register, `FP`, or `SP` register.
- Dreg_even denotes `R0`, `R2`, `R4`, or `R6`.
- Dreg_odd denotes `R1`, `R3`, `R5`, or `R7`.
- DPreg denotes any Data Register File register or any Pointer register, `FP`, or `SP` register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- Ax.L denotes the lower 16 bits of Accumulator `A0.W` or `A1.W`.
- Ax.H denotes the upper 16 bits of Accumulator `A0.W` or `A1.W`.
- Dreg_byte denotes the low order 8 bits of each Data register.
- Option (X) denotes sign extended.
- Option (Z) denotes zero extended.
- * Indicates the status bit may be set or cleared, depending on the result of the instruction.
- ** Indicates the status bit is cleared.
- - Indicates no effect.

Table 2-1: Register File Instructions and Status

Instruction	ASTAT Status Bits						
	AZ	**AN**	**AC0 AC0_COPY AC1**	**AV0 AVS**	**AV1 AV1S**	**CC**	**V V_COPY VS**
allreg = allreg ; [1]	-	-	-	-	-	-	-

Table 2-1: Register File Instructions and Status (Continued)

Instruction	ASTAT Status Bits						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AVS	AV1 AV1S	CC	V V_COPY VS
Ax = Ax ;	-	-	-	-	-	-	-
Ax = Dreg ;	-	-	-	-	-	-	-
Ax = Dreg (X) ;	-	-	-	-	-	-	-
Ax = Dreg (Z) ;	-	-	-	-	-	-	-
A1 = Dreg (X), A0 = Dreg (X) ;	-	-	-	-	-	-	-
A1 = Dreg (Z), A0 = Dreg (Z) ;	-	-	-	-	-	-	-
A1 = Dreg (X), A0 = Dreg (Z) ;	-	-	-	-	-	-	-
A1 = Dreg (Z), A0 = Dreg (X) ;	-	-	-	-	-	-	-
Dreg_even = A0 ;	*	*	-	-	-	-	*
Dreg_odd = A1 ;	*	*	-	-	-	-	*
Dreg_even = A0, Dreg_odd = A1 ;	*	*	-	-	-	-	*
Dreg_odd = A1, Dreg_even = A0 ;	*	*	-	-	-	-	*
IF CC DPreg = DPreg ;	-	-	-	-	-	-	-
IF ! CC DPreg = DPreg ;	-	-	-	-	-	-	-
Dreg = Dreg_lo (Z) ;	*	**	**	-	-	-	**/-
Dreg = Dreg_lo (X) ;	*	*	**	-	-	-	**/-
Ax.X = Dreg_lo ;	-	-	-	-	-	-	-
Dreg_lo = Ax.X ;	-	-	-	-	-	-	-
Ax.L = Dreg_lo ;	-	-	-	-	-	-	-
Ax.H = Dreg_hi ;	-	-	-	-	-	-	-
Dreg_lo = A0 ;	*	*	-	-	-	-	*
Dreg_hi = A1 ;	*	*	-	-	-	-	*
Dreg_hi = A1 ; Dreg_lo = A0 ;	*	*	-	-	-	-	*
Dreg_lo = A0 ; Dreg_hi = A1 ;	*	*	-	-	-	-	*
Dreg = Dreg_byte (Z) ;	*	**	**	-	-	-	**/-
Dreg = Dreg_byte (X) ;	*	*	**	-	-	-	**/-

1. Warning: Not all register combinations are allowed. For details, see the functional description of the Move Register instruction.

Data Types

The processor supports 32-bit words, 16-bit half words, and bytes. The 32- and 16-bit words can be integer or fractional, but bytes are always integers. Integer data types can be signed or unsigned, but fractional data types are always signed. 32-bit words can also be complex numbers constructed from 16-bit real and imaginary parts with the real part in the least significant bits and imaginary part in the most significant bits.

The **Data Types and Representation in Memory/Register** table illustrates the formats for data that resides in memory, in the register file, and in the accumulators. In the table, the letter d represents one bit, and the letter s represents one signed bit.

Some instructions manipulate data in the registers by sign-extending or zero-extending the data to 32 bits:

- Instructions zero-extend unsigned data
- Instructions sign-extend signed 16-bit half words and 8-bit bytes

Other instructions manipulate data as 32-bit numbers. In addition, two 16-bit half words or four 8-bit bytes can be manipulated as 32-bit values.

In the table, note the meaning of these symbols:

- s = sign bit(s)
- d = data bit(s)
- "." = decimal point by convention; however, a decimal point does not literally appear in the number.
- Italics denotes data from a source other than adjacent bits.

Table 2-2: Data Types and Representation in Memory/Register

Format	Representation in Memory	Representation in 32-bit Register
32.0 Unsigned Word	dddd dddd dddd dddd dddd dddd dddd dddd	dddd dddd dddd dddd dddd dddd dddd dddd
32.0 Signed Word	sddd dddd dddd dddd dddd dddd dddd dddd	sddd dddd dddd dddd dddd dddd dddd dddd
16.0 Unsigned Half Word	dddd dddd dddd dddd	0000 0000 0000 0000 dddd dddd dddd dddd
16.0 Signed Half Word	sddd dddd dddd dddd	ssss ssss ssss ssss sddd dddd dddd dddd
8.0 Unsigned Byte	dddd dddd	0000 0000 0000 0000 0000 0000 dddd dddd
8.0 Signed Byte	sddd dddd	ssss ssss ssss ssss ssss ssss sddd dddd
1.15 Signed Fraction	s.ddd dddd dddd dddd	ssss ssss ssss ssss s.ddd dddd dddd dddd
1.31 Signed Fraction	s.ddd dddd dddd dddd dddd dddd dddd dddd	s.ddd dddd dddd dddd dddd dddd dddd dddd
Fractional Complex	s.ddd dddd dddd dddd s.ddd dddd dddd dddd	s.ddd dddd dddd dddd s.ddd dddd dddd dddd
Integer Complex	sddd dddd dddd dddd sddd dddd dddd dddd	sddd dddd dddd dddd sddd dddd dddd dddd
Packed 8.0 Unsigned Byte	dddd dddd dddd dddd dddd dddd dddd dddd	dddd dddd dddd dddd dddd dddd dddd dddd
Packed 1.15 Signed Fraction	s.ddd dddd dddd dddd s.ddd dddd dddd dddd	s.ddd dddd dddd dddd s.ddd dddd dddd dddd

Endianess

Both internal and external memory are accessed in little endian byte order. For more information, see the memory transaction model.

ALU Data Types

Operations on each ALU treat operands and results as either 16- or 32-bit binary strings, except the signed division primitive (DIVS). ALU result status bits treat the results as signed, indicating status with the overflow status bits (AV0, AV1) and the negative status bit (AN). Each ALU has its own sticky overflow status bit, AV0S and AV1S. Once set, these bits remain set until cleared by writing directly to the ASTAT register. An additional V status bit is set or cleared depending on the transfer of the result from both accumulators to the register file. Furthermore, the sticky VS bit is set with the V bit and remains set until cleared.

The logic of the overflow bits (V, VS, AV0, AV0S, AV1, AV1S) is based on two's-complement arithmetic. A bit or set of bits is set if the Most Significant Bit (MSB) changes in a manner not predicted by the signs of the operands and the nature of the operation. For example, adding two positive numbers must generate a positive result; a change in the sign bit signifies an overflow and sets AVx, the corresponding overflow status bits. Adding a negative and a positive number may result in either a negative or positive result, but cannot cause an overflow.

The logic of the carry bits (AC0, AC1) is based on unsigned magnitude arithmetic. The bit is set if a carry is generated from bit 16 (the MSB). The carry bits (AC0, AC1) are most useful for the lower word portions of a multi-word operation.

ALU results generate status information. For more information about using ALU status, see Using Computational Status.

Multiplier Data Types

Each multiplier produces results that are binary strings. The inputs are interpreted according to the information given in the instruction itself (whether it is signed multiplied by signed, unsigned multiplied by unsigned, a mixture, or a rounding operation). The result from the multipliers is either signed or unsigned depending on the sign of the operands and is accordingly zero- or sign-extended to the width of the accumulator.

The 32-bit multiplier multiplies 32-bit operands to produce a 64-bit result. This result is zero- or sign-extended to 72-bits and added to the 72-bit value in A1:0. Two 32-bit complex operands can be multiplied to produce a 64-bit result consisting of a 32-bit real part and 32-bit imaginary part. Both parts are sign extended to 40-bits and the real part added to the value in A0 and the imaginary part added to the value in A1.

The 16-bit multipliers multiply 16-bit operands to produce a 32-bit result which is zero- or sign-extended across the full 40-bit width of the A0 or A1 registers.

The processor supports two modes of format adjustment: the fractional mode for signed fractional operands (1.31 format with 1 sign bit and 31 fractional bits, or 1.15 format with 1 sign bit and 15 fractional bits) and the regular mode for all other combinations operands.

When the processor multiplies two 1.15 operands, the result is a 2.30 (2 sign bits and 30 fractional bits) number. In the fractional mode, the multiplier automatically shifts the multiplier product left one bit before transferring the result to the multiplier result register. This shift of the redundant sign bit causes the multiplier result to be in 1.31 format, which can be rounded to 1.15 format. Similarly when the processor multiplies two 1.31 operands the result is a 2.62 number which is automatically shifted to produce a 1.63 result before transferring to the result register.

In other modes, the left shift does not occur. For example, if the operands are in the 16.0 format, the 32-bit multiplier result would be in 32.0 format. A left shift is not needed and would change the numerical representation.

The multiplier result may be added or subtracted from the value in an accumulator register (`A0`, `A1` or `A1:0`).

For 16-bit signed fractional operands, the 32-bit product output is format adjusted- sign-extended and shifted one bit to the left-before being added to accumulator `A0` or `A1`. For example, bit 31 of the product lines up with bit 32 of `A0` (which is bit 0 of `A0.X`), and bit 0 of the product lines up with bit 1 of `A0` (which is bit 1 of `A0.W`). The Least Significant Bit (LSB) is zero filled. The fractional multiplier result format appears in the **16-bit Signed Fractional Multiplier Results Format** figure.

For integer and unsigned fraction16-bit operands, the 32-bit product register is not shifted before being added to `A0` or `A1`. The **Other 16-bit Multiplier Results Format** figure shows the integer mode result placement.

For 32-bit signed fractional operands, the 64-bit product output is format adjusted- sign-extended and shifted one bit to the left-before being added to accumulator `A1:0`. Bit 63 of the product lines up with bit 64 of `A1:0` (which is bit 0 of `A1.X`), and bit 0 of the product lines up with bit 1 of `A1:0` (which is bit 1 of `A0.W`). The Least Significant Bit (LSB) is zero filled. Note`A0.X`is not used when the combined register `A1:0`holds a 72-bit accumulation result of 32-bit operands.

For other 32-bit integer and unsigned fractional multiplier operands, the 64-bit product is not shifted before being added to `A1:0`.

The result of multiplying 32-bit complex operands, consisting of a 16-bit imaginary part and a 16-bit real part, is a pair of 40-bit signed fractions or signed integers. The accumulation proceeds as for fractional or integer multiplication with the real part of the accumulation performed in `A0` and the imaginary part in `A1`.

Multiplier results generate status information when they update accumulators or when they are transferred to a destination register in the register file. For more information, see Using Computational Status.

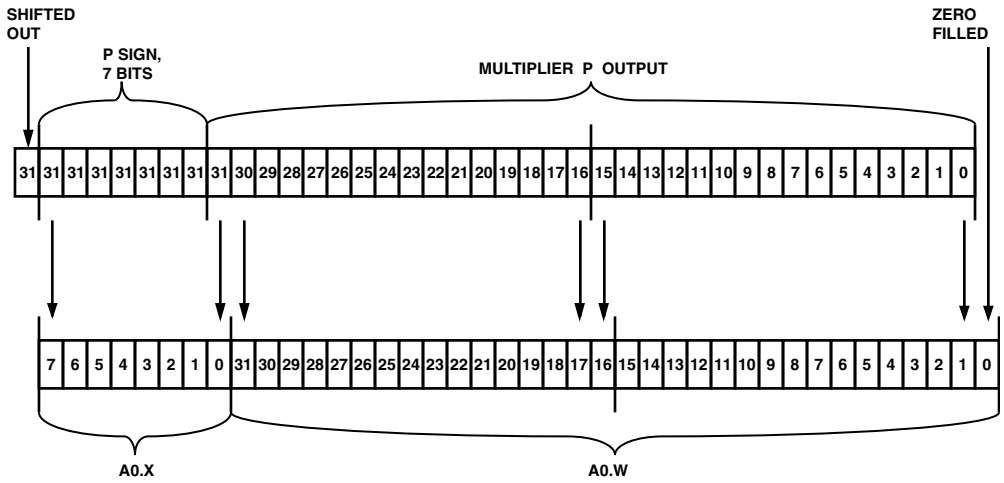


Figure 2-5: 16-bit Signed Fractional Multiplier Results Format

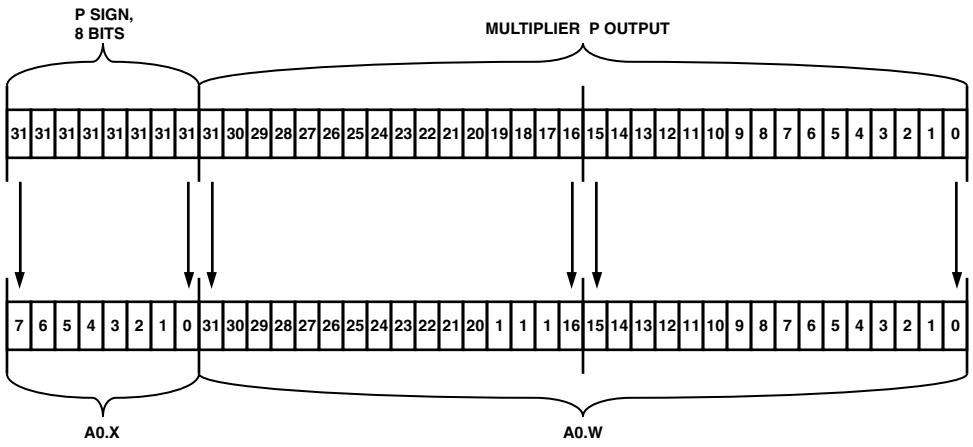


Figure 2-6: Other 16-bit Multiplier Results Format

Shifter Data Types

Many operations in the shifter are explicitly geared to signed (two's-complement) or unsigned values-logical shifts assume unsigned magnitude or binary string values, and arithmetic shifts assume two's-complement values.

The exponent logic assumes two's-complement numbers. The exponent logic supports block floating point, which is also based on two's-complement fractions.

Shifter results generate status information. For more information about using shifter status, see Using Computational Status.

Arithmetic Formats Summary

The **ALU Arithmetic Formats** table, **Multiplier Fractional Modes Formats** table, **Multiplier Arithmetic Integer Modes Formats** table, and **Shifter Arithmetic Formats** table summarize some of the arithmetic characteristics of computational operations.

Table 2-3: ALU Arithmetic Formats

Operation	Operand Formats	Result Formats
Addition	Signed or unsigned	Interpret status bits
Subtraction	Signed or unsigned	Interpret status bits
Logical	Binary string	Same as operands
Division	Explicitly signed or unsigned	Same as operands

Table 2-4: Multiplier Fractional Modes Formats

Operation	Operand Formats	Result Formats
Multiplication	1.15 signed fractional 1.31 signed fractional 0.16 unsigned fractional 0.32 unsigned fractional 2x 1.15 signed fractional complex	2.30 shifted to 1.31 2.62 shifted to 1.63 0.32 not shifted 0.64 not shifted 2x 2.30 shifted to 2x 1.31
Multiplication/Addition	1.15 and 8.32 signed fractional 1.31 and 8.64 signed fractional 0.16 and 8.32 unsigned fractional 0.31 and 8.64 unsigned fractional 2x 1.15 and 2x 8.32 signed fractional complex	8.32 signed 8.64 signed 8.32 unsigned 8.64 unsigned 2x 8.32
Multiplication/Subtraction	1.15 and 8.32 signed fractional 1.31 and 8.64 signed fractional 0.16 and 8.32 unsigned fractional 0.31 and 8.64 unsigned fractional 2x 1.15 and 2x 8.32 signed fractional complex	8.32 signed 8.64 signed 8.32 unsigned 8.64 unsigned 2x 8.32

Table 2-5: Multiplier Arithmetic Integer Modes Formats

Operation	Operand Formats	Result Formats
Multiplication	16.0 explicitly signed or unsigned 32.0 explicitly signed or unsigned 2x 16.0 signed integer complex	32.0 not shifted 64.0 not shifted 2x 64.0 not shifted
Multiplication/Addition	16.0 explicitly signed or unsigned and 40.0 32.0 explicitly signed or unsigned and 72.0 2x 16.0 and 2x 40.0 signed integer complex	40.0 72.0 2x 40.0

Table 2-5: Multiplier Arithmetic Integer Modes Formats (Continued)

Operation	Operand Formats	Result Formats
Multiplication/Subtraction	16.0 explicitly signed or unsigned and 40.0 32.0 explicitly signed or unsigned and 72.0 2x 16.0 and 2x 40.0 signed integer complex	40.0 72.0 2x 40.0

Table 2-6: Shifter Arithmetic Formats

Operation	Operand Formats	Result Formats
Logical Shift	Unsigned binary string	Same as operands
Arithmetic Shift	Signed	Same as operands
Exponent Detect	Signed	Same as operands

Rounding Multiplier Results

On many multiplier operations, the processor supports multiplier results rounding (RND option). Rounding is a means of reducing the precision of a number by removing a lower order range of bits from that number's representation and possibly modifying the remaining portion of the number to more accurately represent its former value. For example, the original number will have N bits of precision, whereas the new number will have only M bits of precision (where N>M). The process of rounding, then, removes N - M bits of precision from the number.

The RND_MOD bit in the ASTAT register determines whether the RND option provides biased or unbiased rounding. For unbiased rounding, set RND_MOD bit = 0. For biased rounding, set RND_MOD bit = 1.

Unbiased Rounding

The convergent rounding method returns the number closest to the original. In cases where the original number lies exactly halfway between two numbers, this method returns the nearest even number, the one containing an LSB of 0. For example, when rounding the 3-bit, two's-complement fraction 0.25 (binary 0. 01) to the nearest 2-bit, two's-complement fraction, the result would be 0.0, because that is the even-numbered choice of 0.5 and 0.0. Since it rounds up and down based on the surrounding values, this method is called unbiased rounding.

Unbiased rounding uses the ALU's capability of rounding 72-bit results at the boundary between bit 31 and bit 32, and of rounding 40-bit results at the boundary between bit 15 and bit 16. Rounding can be specified as part of the instruction code. When rounding is selected, the output register contains the rounded 16-bit result; the accumulator is never rounded.

The accumulator uses an unbiased rounding scheme. The conventional method of biased rounding adds a 1 into bit position 31 or 15 of the adder chain. This method causes a net positive bias because the midway value is always rounded upward.

The accumulator eliminates this bias by forcing bit 32 or bit 16 in the result output to 0 when it detects this midway point. Forcing this bit to 0 has the effect of rounding odd values in the discarded part of the result upward and even values downward, yielding a large sample bias of 0, assuming uniformly distributed values.

The following examples use x to represent any bit pattern (not all zeros). The example in the **Unbiased Multiplier Rounding** figure shows a typical rounding operation for `A0`; the example also applies for `A1`.

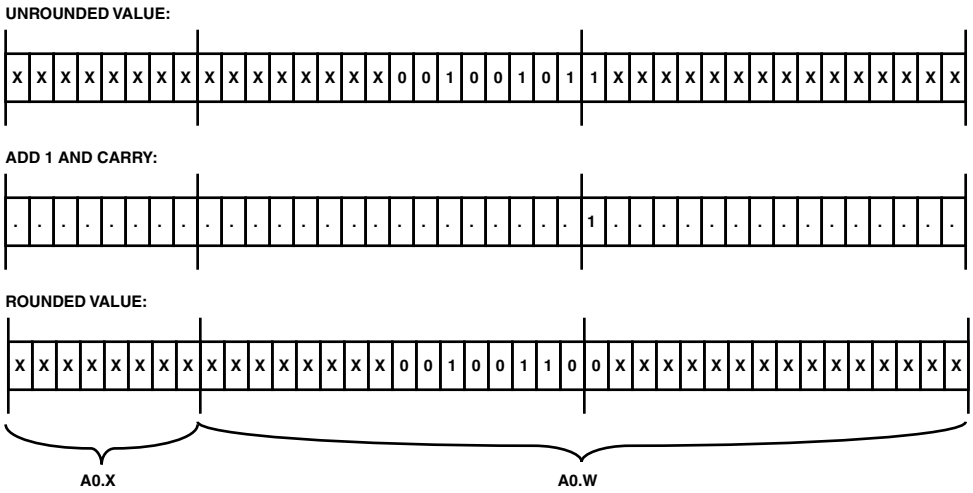


Figure 2-7: Typical Unbiased Multiplier Rounding

The compensation to avoid net bias becomes visible when all lower 15 bits are 0 and bit 15 is 1 (the midpoint value) as shown in the **Avoiding Net Bias in Unbiased Multiplier Rounding** figure. In this figure, `A0` bit 16 is forced to 0. This algorithm is employed on every rounding operation, but is evident only when the bit patterns shown in the lower 16 bits of the next example are present. When a 72-bit value is rounded the bias becomes visible when all lower 31 bits are 0 and bit 31 is 1. In this case, net bias is avoided by forcing bit 32 to 0.

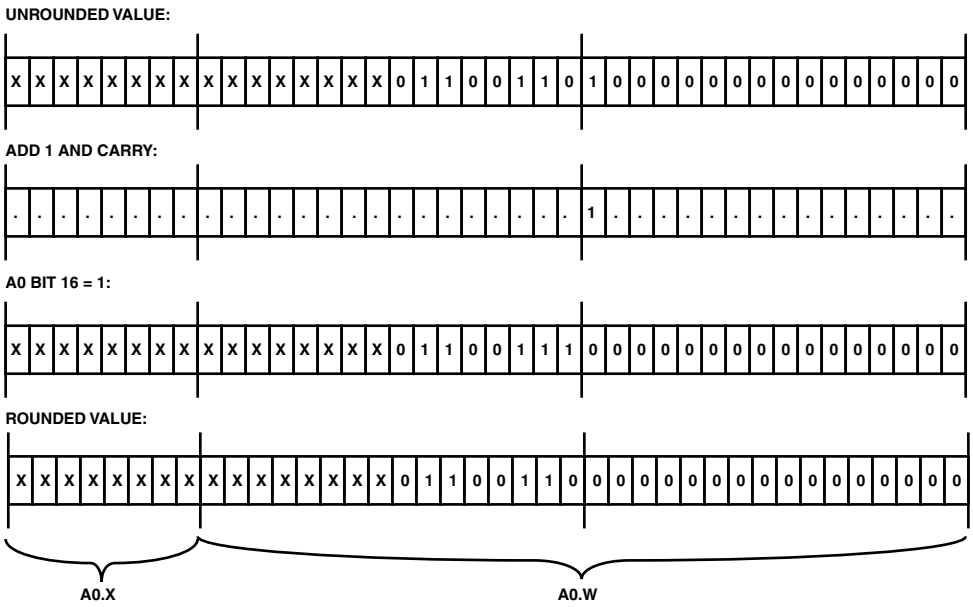


Figure 2-8: Avoiding Net Bias in Unbiased Multiplier Rounding

Biased Rounding

The round-to-nearest method also returns the number closest to the original. However, by convention, an original number lying exactly halfway between two numbers always rounds up to the larger of the two. For example, when rounding the 3-bit, two's-complement fraction 0.25 (binary 0.01) to the nearest 2-bit, two's-complement fraction, this method returns 0.5 (binary 0.1). The original fraction lies exactly midway between 0.5 and 0.0 (binary 0.0), so this method rounds up. Because it always rounds up, this method is called biased rounding.

The `RND_MOD` bit in the `ASTAT` register enables biased rounding. When the `RND_MOD` bit is cleared, the `RND` option in multiplier instructions uses the normal, unbiased rounding operation, as discussed in Unbiased Rounding.

When the `RND_MOD` bit is set (=1), the processor uses biased rounding instead of unbiased rounding. When operating in biased rounding mode, all rounding operations on 72-bit results with `A0.W` set to 0x80000000 round up, rather than only rounding odd values up. Similarly all rounding operations on 40-bit results with `A0.L/A1.L` set to 0x8000 round up. For an example of biased rounding, see Biased Rounding in Multiplier Operation.

Table 2-7: Biased Rounding in Multiplier Operation

A0/A1 Before RND	Biased RND Result	Unbiased RND Result
0x00 0000 8000	0x00 0001 8000	0x00 0000 0000
0x00 0001 8000	0x00 0002 0000	0x00 0002 0000
0x00 0000 8001	0x00 0001 0001	0x00 0001 0001
0x00 0001 8001	0x00 0002 0001	0x00 0002 0001

Table 2-7: Biased Rounding in Multiplier Operation (Continued)

A0/A1 Before RND	Biased RND Result	Unbiased RND Result
0x00 0000 7FFF	0x00 0000 FFFF	0x00 0000 FFFF
0x00 0001 7FFF	0x00 0001 FFFF	0x00 0001 FFFF

Biased rounding affects 40-bit results only when the `A0.L/A1.L` register contains 0x8000 and 72-bit results only when `A0.W` contains 0x80000000; all other rounding operations work normally. This mode allows more efficient implementation of bit specified algorithms that use biased rounding (for example, the Global System for Mobile Communications (GSM) speech compression routines).

Truncation

Another common way to reduce the significant bits representing a number is to simply mask off the N - M lower bits. This process is known as *truncation* and results in a relatively large bias. Instructions that do not support rounding revert to truncation. The `RND_MOD` bit in `ASTAT` has no effect on truncation.

Special Rounding Instructions

The ALU provides the ability to round the arithmetic results directly into a data register with biased or unbiased rounding as described above. It also provides the ability to round on different bit boundaries. The options `RND12`, `RND`, and `RND20` round at bit 12, bit 16, and bit 20, respectively, regardless of the state of the `RND_MOD` bit in `ASTAT`.

For example:

```
R3.L = R4 (RND) ;
```

performs biased rounding at bit 16, depositing the result in a half word.

```
R3.L = R4 + R5 (RND12) ;
```

performs an addition of two 32-bit numbers, biased rounding at bit 12, depositing the result in a half word.

```
R3.L = R4 + R5 (RND20) ;
```

performs an addition of two 32-bit numbers, biased rounding at bit 20, depositing the result in a half word.

Using Computational Status

The multiplier, ALU, and shifter update the overflow and other status bits in the processor's Arithmetic Status (`ASTAT`) register. To use status conditions from computations in program sequencing, use conditional instructions to test the `CC` status bit in the `ASTAT` register after the instruction executes. This method permits monitoring each instruction's outcome. The `ASTAT` register is a 32-bit register, with some bits reserved. To ensure compatibility with future implementations, writes to this register should write back the values read from these reserved bits.

ASTAT Register

The **arithmetic status register** (ASTAT) provides information about the result of an operation. The processor updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation. For more information, see Arithmetic Status Register

Arithmetic Logic Unit (ALU)

The two ALUs perform arithmetic and logical operations on fixed-point data. ALU fixed-point instructions operate on 16-, 32-, and 40-bit fixed-point operands and output 16-, 32-, or 40-bit fixed-point results. ALU instructions include:

- Fixed-point addition and subtraction of registers
- Addition and subtraction of immediate values
- Accumulation and subtraction of multiplier results
- Logical AND, OR, NOT, XOR, bitwise XOR, Negate
- Functions: ABS, MAX, MIN, Round, division primitives

ALU Operations

Primary ALU operations occur on ALU0, while parallel operations occur on ALU1, which performs a subset of ALU0 operations.

The **Inputs and Outputs of Each ALU** table describes the possible inputs and outputs of each ALU.

Table 2-8: Inputs and Outputs of Each ALU

Input	Output
Two or four 16-bit operands	One or two 16-bit results
Two 32-bit operands	One 32-bit result
32-bit result from the multiplier	Combination of 32-bit result from the multiplier with a 40-bit accumulation result

Combining operations in both ALUs can result in four 16-bit results, two 32-bit results, or two 40-bit results generated in a single instruction.

Single 16-Bit Operations

In single 16-bit operations, any two 16-bit register halves may be used as the input to the ALU. An addition, subtraction, or logical operation produces a 16-bit result that is deposited into an arbitrary destination register half. ALU0 is used for this operation, because it is the primary resource for ALU operations.

For example:

```
R3.H = R1.H + R2.L (NS) ;
```

adds the 16-bit contents of `R1.H` (`R1` high half) to the contents of `R2.L` (`R2` low half) and deposits the result in `R3.H` (`R3` high half) with no saturation.

Dual 16-Bit Operations

In dual 16-bit operations, any two 32-bit registers may be used as the input to the ALU, considered as pairs of 16-bit operands. An addition, subtraction, or logical operation produces two 16-bit results that are deposited into an arbitrary 32-bit destination register. ALU0 is used for this operation, because it is the primary resource for ALU operations.

For example:

```
R3 = R1 +|- R2 (S) ;
```

adds the 16-bit contents of `R2.H` (`R2` high half) to the contents of `R1.H` (`R1` high half) and deposits the result in `R3.H` (`R3` high half) with saturation.

The instruction also subtracts the 16-bit contents of `R2.L` (`R2` low half) from the contents of `R1.L` (`R1` low half) and deposits the result in `R3.L` (`R3` low half) with saturation (see 16-bit Multiplier Data Flow Details).

Quad 16-Bit Operations

In quad 16-bit operations, any two 32-bit registers may be used as the inputs to ALU0 and ALU1, considered as pairs of 16-bit operands. A small number of addition or subtraction operations produces four 16-bit results that are deposited into two arbitrary, 32-bit destination registers. Both ALU0 and ALU1 are used for this operation. Because there are only two 32-bit data paths from the Data Register File to the arithmetic units, the same two pairs of 16-bit inputs are presented to ALU1 as to ALU0. The instruction construct is identical to that of a dual 16-bit operation, and input operands must be the same for both ALUs.

For example:

```
R3 = R0 +|+ R1, R2 = R0 -|- R1 (S) ;
```

performs four operations:

- Adds the 16-bit contents of `R1.H` (`R1` high half) to the 16-bit contents of `R0.H(R0` high half) and deposits the result in `R3.H` with saturation.
- Adds `R1.L` to `R0.L` and deposits the result in `R3.L` with saturation.
- Subtracts the 16-bit contents of `R1.H` (`R1` high half) from the 16-bit contents of the `R0.H(R0` high half) and deposits the result in `R2.H` with saturation.
- Subtracts `R1.L` from `R0.L` and deposits the result in `R2.L` with saturation.

Explicitly, the four equivalent instructions are:

```
R3.H = R0.H + R1.H (S) ;
R3.L = R0.L + R1.L (S) ;
R2.H = R0.H - R1.H (S) ;
R2.L = R0.L - R1.L (S) ;
```

Single 32-Bit Operations

In single 32-bit operations, any two 32-bit registers may be used as the input to the ALU, considered as 32-bit operands. An addition, subtraction, or logical operation produces a 32-bit result that is deposited into an arbitrary 32-bit destination register. ALU0 is used for this operation, because it is the primary resource for ALU operations.

In addition to the 32-bit input operands coming from the Data Register File, operands may be sourced and deposited into the Pointer Register File, consisting of the eight registers `P[5:0]`,`SP`,`FP`.

For example:

```
R3 = R1 + R2 (NS) ;
```

adds the 32-bit contents of `R2` to the 32-bit contents of `R1` and deposits the result in `R3` with no saturation.

```
R3 = R1 + R2 (S) ;
```

adds the 32-bit contents of `R1` to the 32-bit contents of `R2` and deposits the result in `R3` with saturation.

Dual 32-Bit Operations

In dual 32-bit operations, any two 32-bit registers may be used as the input to ALU0 and ALU1, considered as a pair of 32-bit operands. An addition or subtraction produces two 32-bit results that are deposited into two 32-bit destination registers. Both ALU0 and ALU1 are used for this operation. Because only two 32-bit data paths go from the Data Register File to the arithmetic units, the same two 32-bit input registers are presented to ALU0 and ALU1.

For example:

```
R3 = R1 + R2, R4 = R1 - R2 (NS);
```

adds the 32-bit contents of `R2` to the 32-bit contents of `R1` and deposits the result in `R3` with no saturation.

The instruction also subtracts the 32-bit contents of `R2` from that of `R1` and deposits the result in `R4` with no saturation.

A specialized form of this instruction uses the ALU 40-bit result registers as input operands, creating the sum and differences of the `A0` and `A1` registers.

For example:

```
R3 = A0 + A1, R4 = A0 - A1 (S);
```

transfers to the result registers two 32-bit, saturated, sum and difference values of the ALU registers.

ALU Division Support Features

The ALU supports division with two special divide primitives. These instructions (`DIVS`, `DIVQ`) let programs implement a non-restoring, conditional (error checking), addition/subtraction/division algorithm.

The division can be either signed or unsigned, but both the dividend and divisor must be of the same type. Details about using division and programming examples are available in the arithmetic operation chapter.

Special SIMD Video ALU Operations

Four 8-bit Video ALUs enable the processor to process video information with high efficiency. Each Video ALU instruction may take from one to four pairs of 8-bit inputs and return one to four 8-bit results. The inputs are presented to the Video ALUs in two 32-bit words from the Data Register File. The possible operations include:

- Quad 8-Bit Add or Subtract
- Quad 8-Bit Average
- Quad 8-Bit Pack or Unpack
- Quad 8-Bit Subtract-Absolute-Accumulate
- Byte Align

For more information about the operation of these instructions, see the video/pixel operation instructions chapter.

ALU Instruction Summary

The ALU Instructions and Status table lists the ALU instructions. For more information about assembly language syntax and the effect of ALU instructions on the status bits, see ALU Instructions and Status.

In the table, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo_hi denotes any 16-bit register half in any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- imm7 denotes a signed, 7-bit wide, immediate value.

- A*x* denotes either ALU Result register `A0` or `A1`.
- DIVS denotes a Divide Sign primitive.
- DIVQ denotes a Divide Quotient primitive.
- MAX denotes the maximum, or most positive, value of the source registers.
- MIN denotes the minimum value of the source registers.
- ABS denotes the absolute value of the upper and lower halves of a single 32-bit register.
- RND denotes rounding a half word.
- RND12 denotes saturating the result of an addition or subtraction and rounding the result on bit 12.
- RND20 denotes saturating the result of an addition or subtraction and rounding the result on bit 20.
- SIGNBITS denotes the number of sign bits in a number, minus one.
- EXPADJ denotes the lesser of the number of sign bits in a number minus one, and a threshold value.
- * Indicates the status bit may be set or cleared, depending on the results of the instruction.
- ** Indicates the status bit is cleared.
- - Indicates no effect.
- *d* indicates `AQ` contains the dividend MSB Exclusive-OR divisor MSB.

Table 2-9: ALU Instructions and Status

Instruction	ASTAT Status Bits						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Dreg = Dreg + Dreg ;	*	*	*	-	-	*	-
Dreg = Dreg - Dreg (S) ;	*	*	*	-	-	*	-
Dreg = Dreg + Dreg, Dreg = Dreg - Dreg ;	*	*	*	-	-	*	-
Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi ;	*	*	*	-	-	*	-
Dreg_lo_hi = Dreg_lo_hi - Dreg_lo_hi (S) ;	*	*	*	-	-	*	-
Dreg = Dreg +|+ Dreg ;	*	*	*	-	-	*	-
Dreg = Dreg +|- Dreg ;	*	*	*	-	-	*	-
Dreg = Dreg -|+ Dreg ;	*	*	*	-	-	*	-
Dreg = Dreg -|- Dreg ;	*	*	*	-	-	*	-

Table 2-9: ALU Instructions and Status (Continued)

Instruction	ASTAT Status Bits						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Dreg = Dreg +|+Dreg, Dreg = Dreg -|- Dreg ;	*	*	-	-	-	*	-
Dreg = Dreg +|- Dreg, Dreg = Dreg -|+ Dreg ;	*	*	-	-	-	*	-
Dreg = A*x* + A*x*, Dreg = A*x* - A*x* ;	*	*	*	-	-	*	-
Dreg += imm7 ;	*	*	*	-	-	*	-
Dreg = (A0 += A1) ;	*	*	*	*	-	*	-
Dreg_lo_hi = (A0 += A1) ;	*	*	*	*	-	*	-
A0 += A1 ;	*	*	*	*	-	-	-
A0 -= A1 ;	*	*	*	*	-	-	-
DIVS (Dreg, Dreg) ;	*	*	*	*	-	-	d
DIVQ (Dreg, Dreg) ;	*	*	*	*	-	-	d
Dreg = MAX (Dreg, Dreg) (V) ;	*	*	-	-	-	**/-	-
Dreg = MIN (Dreg, Dreg) (V) ;	*	*	-	-	-	**/-	-
Dreg = ABS Dreg (V) ;	*	**	-	-	-	*	-
A*x* = ABS A*x* ;	*	**	-	*	*	*	-
A*x* = ABS A*x*, A*x* = ABS A*x* ;	*	**	-	*	*	*	-
A*x* = -A*x* ;	*	*	*	*	*	*	-
A*x* = -A*x*, A*x* =- A*x* ;	*	*	*	*	*	*	-
A*x* = A*x* (S) ;	*	*	-	*	*	-	-
A*x* = A*x* (S), A*x* = A*x* (S) ;	*	*	-	*	*	-	-
Dreg_lo_hi = Dreg (RND) ;	*	*	-	-	-	*	-
Dreg_lo_hi = Dreg + Dreg (RND12) ;	*	*	-	-	-	*	-
Dreg_lo_hi = Dreg - Dreg (RND12) ;	*	*	-	-	-	*	-
Dreg_lo_hi = Dreg + Dreg (RND20) ;	*	*	-	-	-	*	-
Dreg_lo_hi = Dreg - Dreg (RND20) ;	*	*	-	-	-	*	-
Dreg_lo = SIGNBITS Dreg ;	-	-	-	-	-	-	-
Dreg_lo = SIGNBITS Dreg_lo_hi ;	-	-	-	-	-	-	-
Dreg_lo = SIGNBITS An ;	-	-	-	-	-	-	-

Table 2-9: ALU Instructions and Status (Continued)

Instruction	ASTAT Status Bits						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Dreg_lo = EXPADJ (Dreg, Dreg_lo) (V) ;	-	-	-	-	-	-	-
Dreg_lo = EXPADJ (Dreg_lo_hi, Dreg_lo);	-	-	-	-	-	-	-
Dreg = Dreg & Dreg ;	*	*	**	-	-	**/-	-
Dreg = ~ Dreg ;	*	*	**	-	-	**/-	-
Dreg = Dreg | Dreg ;	*	*	**	-	-	**/-	-
Dreg = Dreg ^ Dreg ;	*	*	**	-	-	**/-	-
Dreg =- Dreg ;	*	*	*	-	-	*	-

Multiply Accumulators (Multipliers)

The three multipliers (MAC10, MAC0 and MAC1) perform fixed-point multiplication and multiply and accumulate operations. Multiply and accumulate operations are available with either cumulative addition or cumulative subtraction.

MAC10 executes fixed-point instructions which operate on 32-bit fixed-point data and produce 64-bit results that may be added or subtracted from a 72-bit accumulator. It also executes instructions which operate on 32-bit complex fixed-point data and produce 64-bit results that may be added or subtracted from an 80-bit accumulator. MAC0 and MAC1 execute fixed-point instructions which operate on 16-bit fixed-point data and produce 32-bit results that may be added or subtracted from a 40-bit accumulator.

Inputs are treated as fractional, fractional complex, integer or integer complex, unsigned or two's-complement. Multiplier instructions include:

- Multiplication
- Multiply and accumulate with addition, rounding optional
- Multiply and accumulate with subtraction, rounding optional
- Dual versions of the above operations with 16-bit operands

Multiplier Operation

Each of the 16-bit multipliers, MAC0 and MAC1, has two 32-bit inputs from which it derives the two 16-bit operands. For single multiply and accumulate instructions, these operands can be any Data registers in the Data Register File. Each multiplier can accumulate results in its Accumulator register, `A1` or `A0`. The

accumulator results can be saturated to 32 or 40 bits. The multiplier result can also be written directly to a 16- or 32-bit destination register with optional rounding.

The 32-bit multiplier, MAC10, has two 32-bit operands which can be in any Data register in the Data Register File. The multiplier can accumulate results in the register pair A1:0. The accumulator results are saturated to 72 bits. The 64-bit multiplier result can also be written directly to a data register pair or rounded or truncated to 32-bits and written to a Data Register.

Each multiplier instruction determines whether the inputs are either both in integer format or both in fractional format. The format of the result matches the format of the inputs. In MAC0, both inputs are treated as signed or unsigned. In MAC10 and MAC1, there is a mixed-mode option. Complex multiplication instructions, executed by MAC10, determine whether the inputs are either both complex signed integer or complex signed fractional.

If both inputs are fractional and signed, the multiplier automatically shifts the result left one bit to remove the redundant sign bit. Unsigned fractional, integer, and mixed modes do not perform a shift for sign bit correction. Multiplier instruction options specify the data format of the inputs. See Multiplier Instruction Options for more information.

Placing Multiplier Results in Multiplier Accumulator Registers

As shown in 16-bit Multiplier Data Flow Details, each multiplier may write results to dedicated accumulator registers, `A0` or `A1`. MAC0 writes to `A0`, MAC1 writes to `A1` and MAC10 writes to both `A0` and `A1`. Each Accumulator register is divided into three sections- `A0.L/A1.L`(bits 15:0), `A0.H/A1.H` (bits 31:16), and `A0.X/A1.X` (bits 39:32).

When MAC0 or MAC1 writes to its result Accumulator register, the 32-bit result is deposited into the lower bits of the combined Accumulator register, and the MSB is sign-extended into the upper eight bits of the register (`A0.X/A1.X`).

The accumulators are used as a pair, `A1:0`, for the results of 32-bit fixed point and complex multiplication instructions. MAC10 writes the low 32-bits of the 64-bit fixed point result into `A0.W`, the high 32-bits to `A1.W`, and the value is sign, or zero extended into `A1.X`. `A0.X` is always set to zero. MAC10 writes the real part of complex results to `A0` and the imaginary part to `A1`.

The accumulator pair can be initialized by transferring data from a data register pair using a dual register move.

For example, these instructions transfer 64-bit values into the 72-bit accumulator:

```
A1 = R1 (X), A0 = R0 (Z); /* sign extend R1:0 into A1:0 */
A1 = R1 (Z), A0 = R0 (Z); /* zero extend R1:0 into A1:0 */
A1 = A0 = 0; /* A1:0 = 0; */
```

Multiplier output can be deposited not only in the `A1:0`, `A0` or `A1` registers, but also in a variety of 16- or 32-bit Data registers in the Data Register File.

Rounding or Saturating Multiplier Results

On a multiply and accumulate operation, the accumulator data can be saturated and, optionally, rounded for extraction to a register pair, register or register half. When a multiply deposits a result only in a register pair, register or register half, the saturation and rounding works the same way. The rounding and saturation operations work as follows.

- Rounding is applied only to fractional results except for the `IH` option, which applies rounding and high half extraction to an integer result.

  For the `IH` option, the rounded result is obtained by adding 0x8000 to the accumulator (for MAC) or multiply result (for MULT) and then saturating to 32-bits. For more information, see Multiplier Instruction Options.

  Rounding cannot be combined with a multiply and accumulate into the 72-bit accumulator,`A1:0`, but can be performed by an instruction which only extracts the value from`A1:0`into a Data register.

- If an overflow or underflow has occurred, the saturate operation sets the specified Result register to the maximum positive or negative value. For more information, see the following section.

- The `NS` option prevents saturation. With this option, when an overflow or underflow has occurred, the specified Result register is set to the low order bits of the full result. The `NS` option is only supported for integer multiplications of 32-bit operands.

Saturating Multiplier Results on Overflow

The following bits in `ASTAT` indicate multiplier overflow status:

- Bit 16 (`AV0`) and bit 18 (`AV1`) record overflow condition (whether the result has overflowed 32 bits) for the `A0` and `A1` accumulators, respectively. Bit 16 (`AV0`) also records the overflow condition for `A1:0`. If the bit is cleared (=0), no overflow or underflow has occurred. If the bit is set (=1), an overflow or underflow has occurred. The `AV0S` and `AV1S` bits are sticky bits.

- Bit 24 (`V`) and bit 25 (`VS`) are set if overflow occurs in extracting the accumulator result to a register.

32-bit Multiplier Data Flow Details

The 32-bit multiplier has two 32-bit inputs, performs a 32-bit fixed point or complex multiplication, and stores the result in the combined accumulator register, `A1:0`, or extracts to a 32-bit register or 64-bit register pair.

For complex calculations the 32-bit real and imaginary results are passed to 40-bit adder/subtracters which may add or subtract the new result from the values in the Accumulator Registers, with the real part in `A0` and imaginary part in `A1`, or may be written directly to the Data Register File Result register.

For fixed point calculations the 64-bit product is passed to a 72-bit adder/subtracter, which may add or subtract the new product from the contents of the combined Accumulator Result register, `A1:0`, or pass

the new product directly to the Data Register File Results register. The fixed point value in A1:0is 72 bits wide. This register consists of smaller 32- and 8-bit registers A0.W, A1.W, and A1.X. For example:

```
A1:0 += R2 * R3 ;
```

In this instruction, the combined multiplier/accumulator performs a multiply and accumulates the result with the previous results in the A1:0Accumulator.

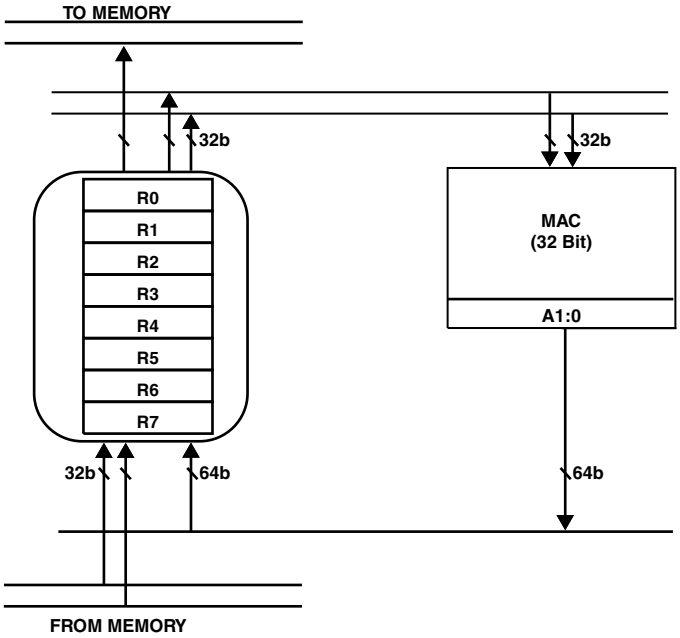


Figure 2-9: 32-bit Multiplier/Accumulators

32-bit Multiply Without Accumulate

The multiplier may operate without the accumulation function. If accumulation is not used, the result can be directly stored in a register from the Data Register File or the Accumulator register. The destination may be a 32-bit register or a 64-bit register pair. If the destination register is 32 bits, then the word that is extracted from the multiplier depends on the data type of the input.

- If the multiplication uses fractional operands, then the high half of the result is extracted and stored in the 32-bit destination register.
- If the multiplication uses integer operands, then the low half of the result is extracted and stored in the 32-bit destination register. These extractions provide the most useful information in the resultant 32-bit word for the data type chosen.
- If the multiplication uses complex fractional operands, then the high half of the real result is extracted and stored in the low half of the 32-bit destination register and the high half of the imaginary result is extracted and stored in the high half of the 32-bit result register.

- If the multiplication uses complex integer operands, then the low half of the real result is extracted and stored in the low half of the 32-bit destination register and the low half of the imaginary result is extracted and stored in the high half of the 32-bit result register.

For example, this instruction uses fractional, unsigned operands:

```
R0 = R1 * R2 (FU) ;
```

The instruction deposits the upper 32 bits of the multiply answer with rounding and saturation into `R0`. This instruction uses unsigned integer operands:

```
R1 = R2 * R3 (IU,NS) ;
```

The instruction deposits the lower 32 bits of the multiply answer with no saturation into `R1`.

```
R1:0 = R1 * R2 ;
```

Regardless of operand type, the preceding operation computes 64 bits of the multiplier answer with saturation, deposits the high 32-bits into `R1` and the low 32-bits into `R0`.

This instruction uses complex fractional operands:

```
R1 = cmul(R2, R3) ;
```

The instruction deposits the higher 16 bits of the real part of the multiply answer in `R1.L`, and the higher 16 bits of the imaginary part of the multiply answer in `R1.H`. This instruction uses complex fractional operands:

```
R1 = cmul(R2, R3)(IS);
```

The instruction deposits the lower 16 bits of the real part of the multiply answer in `R1.L`, and the lower 16 bits of the imaginary part of the multiply answer in `R1.H`.

```
R1:0 = cmul(R2, R3) ;
```

The preceding operation deposits the full 32 bits of the real part of the multiply answer in `R0`, and the full 32 bits of the imaginary part of the multiply answer in `R1`.

The processor supports a two operand version of the 32-bit multiply instruction for backward compatibility:

```
R0 *= R1 ;
```

This is equivalent to:

```
R0 = R0 * R1 (IS,NS) ;
```

16-bit Multiplier Data Flow Details

The **16-bit Multiplier/Accumulators** figure shows the Register file along with the 16-bit multiplier/accumulators.

Each 16-bit multiplier has two 16-bit inputs, performs a 16-bit multiplication, and stores the result in a 40-bit accumulator or extracts to a 16-bit or 32-bit register. Two 32-bit words are available at the MAC inputs, providing four 16-bit operands to chose from.

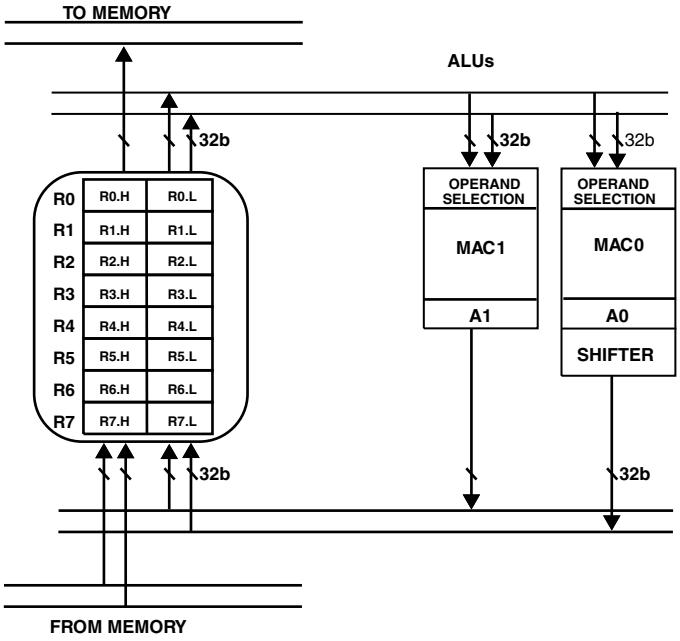


Figure 2-10: 16-bit Multiplier/Accumulators

One of the operands must be selected from the low half or the high half of one 32-bit word. The other operand must be selected from the low half or the high half of the other 32-bit word. Thus, each MAC is presented with four possible input operand combinations. The two 32-bit words can contain the same register information, giving the options for squaring and multiplying the high half and low half of the same register. The **Four Possible Combinations of 16-bit MAC Operands** figure shows these possible combinations.

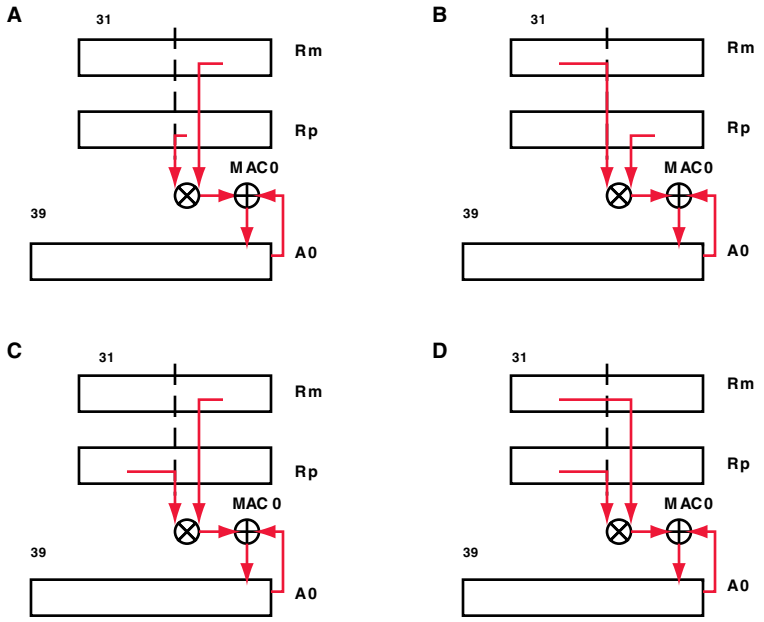


Figure 2-11: Four Possible Combinations of 16-bit MAC Operands

The 32-bit product is passed to a 40-bit adder/subtracter, which may add or subtract the new product from the contents of the Accumulator Result register or pass the new product directly to the Data Register File Results register. For results, the `A0` and `A1` registers are 40 bits wide. Each of these registers consists of smaller 32- and 8-bit registers-`A0.W`, `A1.W`, `A0.X`, and `A1.X`.

For example:

```
A1 += R3.H * R4.H ;
```

In this instruction, the `MAC1` multiplier/accumulator performs a multiply and accumulates the result with the previous results in the `A1` Accumulator.

16-bit Multiply Without Accumulate

The 16-bit multiplier may operate without the accumulation function. If accumulation is not used, the result can be directly stored in a register from the Data Register File or the Accumulator register. The destination register may be 16 bits or 32 bits. If a 16-bit destination register is a low half, then MAC0 is used; if it is a high half, then MAC1 is used. For a 32-bit destination register, either MAC0 or MAC1 is used.

If the destination register is 16 bits, then the word that is extracted from the multiplier depends on the data type of the input.

- If the multiplication uses fractional operands or the `IH` option, then the high half of the result is extracted and stored in the 16-bit destination registers (see the **Multiplication of 16-bit Fractional Operands** figure).
- If the multiplication uses integer operands, then the low half of the result is extracted and stored in the 16-bit destination registers. These extractions provide the most useful information in the resultant 16-bit word for the data type chosen (see the **Multiplication of 16-bit Integer Operands** figure).

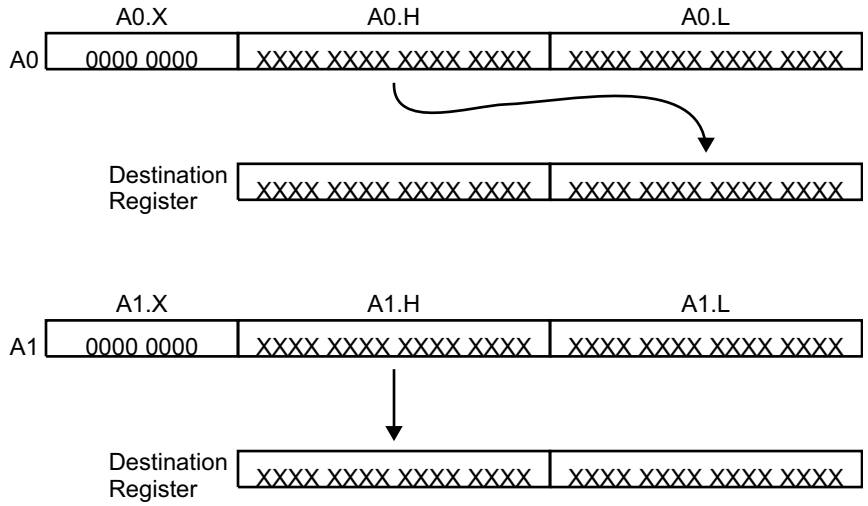


Figure 2-12: Multiplication of 16-bit Fractional Operands

For example, this instruction uses fractional, unsigned operands:

```
R0.L = R1.L * R2.L (FU) ;
```

The instruction deposits the upper 16 bits of the multiply answer with rounding and saturation into the lower half of `R0`, using MAC0. This instruction uses unsigned integer operands:

```
R0.H = R2.H * R3.H (IU) ;
```

The instruction deposits the lower 16 bits of the multiply answer with any required saturation into the high half of `R0`, using MAC1 .

```
R0 = R1.L * R2.L ;
```

Regardless of operand type, the preceding operation deposits 32 bits of the multiplier answer with saturation into `R0`, using MAC0.

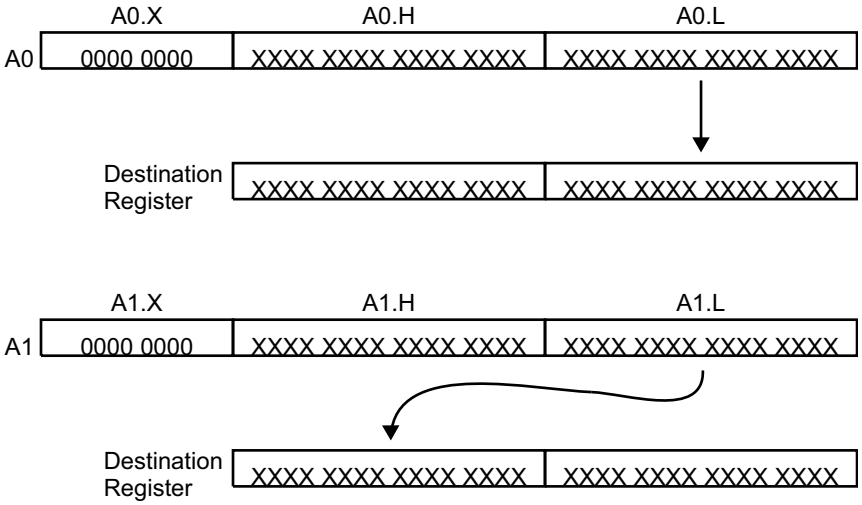


Figure 2-13: Multiplication of 16-bit Integer Operands

Dual 16-bit MAC Operations

The processor has two 16-bit MACs. Both MACs can be used in the same operation to double the MAC throughput. The same two 32-bit input registers are offered to each MAC unit, providing each with four possible combinations of 16-bit input operands. Dual MAC operations are frequently referred to as vector operations, because a program could store vectors of samples in the four input operands and perform vector computations.

An example of a dual multiply and accumulate instruction is

```
A1 += R1.H * R2.L, A0 += R1.L * R2.H ;
```

This instruction represents two multiply and accumulate operations.

- In one operation (MAC1) the high half of `R1` is multiplied by the low half of `R2` and added to the contents of the `A1` Accumulator.
- In the second operation (MAC0) the low half of `R1` is multiplied by the high half of `R2` and added to the contents of `A0`.

The results of the MAC operations may be written to registers in a number of ways: as a pair of 16-bit halves, as a pair of 32-bit registers, or as an independent 16-bit half register or 32-bit register.

For example:

```
R3.H = (A1 += R1.H * R2.L), R3.L = (A0 += R1.L * R2.L) ;
```

In this instruction, the 40-bit Accumulator is packed into a 16-bit half register. The result from MAC1 must be transferred to a high half of a destination register and the result from MAC0 must be transferred to the low half of the same destination register.

The operand type determines the correct bits to extract from the Accumulator and deposit in the 16-bit destination register. See 16-bit Multiply Without Accumulate.

```
R3 = (A1 += R1.H * R2.L), R2= (A0 += R1.L * R2.L) ;
```

In this instruction, the 40-bit Accumulators are packed into two 32-bit registers. The registers must be register pairs (`R[1:0]`, `R[3:2]`, `R[5:4]`, `R[7:6]`).

```
R3.H = (A1 += R1.H * R2.L), A0 += R1.L * R2.L ;
```

This instruction is an example of one Accumulator-but not the other-being transferred to a register. Either a 16- or 32-bit register may be specified as the destination register.

Multiplier Instruction Summary

The Multiplier Instructions and Status table lists the multiplier instructions. For more information about assembly language syntax and the effect of multiplier instructions on the status bits, see Multiplier Instructions and Status.

In the table, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo_hi denotes any 16-bit register half in any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- Dreg_pair denotes a pair of adjacent Data Register File registers. The low half of a 64-bit value is held in the lower numbered, even, register and the high half in the higher numbered, odd, register.
- A x denotes either MAC Accumulator register `A0` or `A1`.
- * Indicates the status bit may be set or cleared, depending on the results of the instruction.
- - Indicates no effect.

Multiplier instruction options are described in Multiplier Instruction Options.

Table 2-10: Multiplier Instructions and Status

Instruction	ASTAT Status Bits		
	AV0 AV0S	AV1 AV1S	V V_COPY VS
Dreg = Dreg * Dreg ;	-	-	*
Dreg_pair = Dreg * Dreg ;	-	-	*
A1:0 = Dreg * Dreg ;	*	-	-

Table 2-10: Multiplier Instructions and Status (Continued)

Instruction	ASTAT Status Bits		
	AV0 AV0S	AV1 AV1S	V V_COPY VS
A1:0 += Dreg * Dreg ;	*	-	-
A1:0 -= Dreg * Dreg ;	*	-	-
Dreg = (A1:0 = Dreg * Dreg) ;	*	-	*
Dreg = (A1:0 += Dreg * Dreg) ;	*	-	*
Dreg = (A1:0 -= Dreg * Dreg) ;	*	-	*
Dreg_pair = (A1:0 = Dreg * Dreg) ;	*	-	*
Dreg_pair = (A1:0 += Dreg * Dreg) ;	*	-	*
Dreg_pair = (A1:0 -= Dreg * Dreg) ;	*	-	*
Dreg = cmul(Dreg, Dreg) ;	-	-	*
Dreg_pair = cmul(Dreg, Dreg) ;	-	-	*
A1:0 = cmul(Dreg, Dreg) ;	*	*	-
A1:0 += cmul(Dreg, Dreg) ;	*	*	-
A1:0 -= cmul(Dreg, Dreg) ;	*	*	-
Dreg = (A1:0 = cmul(Dreg, Dreg)) ;	*	*	*
Dreg = (A1:0 += cmul(Dreg, Dreg)) ;	*	*	*
Dreg = (A1:0 -= cmul(Dreg, Dreg)) ;	*	*	*
Dreg_pair = (A1:0 = cmul(Dreg, Dreg)) ;	*	*	*
Dreg_pair = (A1:0 += cmul(Dreg, Dreg)) ;	*	*	*
Dreg_pair = (A1:0 -= cmul(Dreg, Dreg)) ;	*	*	*
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ;	-	-	*
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi ;	-	-	*
Dreg = Dreg_lo_hi * Dreg_lo_hi ;	-	-	*
A x = Dreg_lo_hi * Dreg_lo_hi ;	*	*	-
A x += Dreg_lo_hi * Dreg_lo_hi ;	*	*	-
An -= Dreg_lo_hi * Dreg_lo_hi ;	*	*	-
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*

Table 2-10: Multiplier Instructions and Status (Continued)

Instruction	ASTAT Status Bits		
	AV0 AV0S	AV1 AV1S	V V_COPY VS
Dreg = (A x = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (A x += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (A x -= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg *= Dreg ;	-	-	-

Multiplier Instruction Options

The following descriptions of multiplier instruction options provide an overview. Not all options are available for all instructions. For information about how to use these options with their respective instructions, see the arithmetic operation instructions chapter.

default

No option; input data is signed fraction.

`(IS)`

Input data operands are signed integer. No shift correction is made.

`(FU)`

Input data operands are unsigned fraction. No shift correction is made.

`(IU)`

Input data operands are unsigned integer. No shift correction is made.

`(T)`

Input data operands are signed fraction. When copying to the destination half register, truncates the lower 16 bits of the Accumulator contents.

`(TFU)`

Input data operands are unsigned fraction. When copying to the destination half register, truncates the lower 16 bits of the Accumulator contents.

`(ISS2)`

If multiplying and accumulating to a register:

Input data operands are signed integer. When copying to the destination register, Accumulator contents are scaled (multiplied x2 by a one-place shift-left). If scaling produces a signed value larger than 32 bits, the number is saturated to its maximum positive or negative value.

If multiplying and accumulating to a half register:

When copying the lower 16 bits to the destination half register, the Accumulator contents are scaled. If scaling produces a signed value greater than 16 bits, the number is saturated to its maximum positive or negative value.

(IH)

This option indicates integer multiplication with high half word extraction. The Accumulator is saturated at 32 bits, and bits [31:16] of the Accumulator are rounded, and then copied into the destination half register.

(W32)

Input data operands are signed fraction with no extension bits in the Accumulators at 32 bits. Left-shift correction of the product is performed, as required. This option is used for legacy GSM speech vocoder algorithms written for 32-bit Accumulators. For this option only, this special case applies: 0x8000 x 0x8000 = 0x7FFF.

(M)

Operation uses mixed-multiply mode. Valid only for MAC1 versions of the instruction. Multiplies a signed fraction by an unsigned fractional operand with no left-shift correction. Operand one is signed; operand two is unsigned. MAC0 performs an unmixed multiply on signed fractions by default, or another format as specified. That is, MAC0 executes the specified signed/signed or unsigned/unsigned multiplication. The (M) option can be used alone or in conjunction with one other format option.

(NS)

Operation is non-saturating. When copying the Accumulator contents to a destination register, the low order bits are copied if the whole value will not fit in the destination. The (NS) option can only be used in conjunction with integer format options.

Barrel Shifter (Shifter)

The shifter provides bitwise shifting functions for 16-, 32-, or 40-bit inputs, yielding a 16-, 32-, or 40-bit output. These functions include arithmetic shift, logical shift, rotate, and various bit test, set, pack, unpack, and exponent detection functions. These shift functions can be combined to implement numerical format control, including full floating-point representation.

Shifter Operations

The shifter instructions (>>>, >>, <<, ASHIFT, LSHIFT, ROT) can be used various ways, depending on the underlying arithmetic requirements. The ASHIFT and >>> instructions represent the arithmetic shift. The LSHIFT, <<, and >> instructions represent the logical shift.

The arithmetic shift and logical shift operations can be further broken into subsections. Instructions that are intended to operate on 16-bit single or paired numeric values (as would occur in many DSP algorithms) can use the instructions ASHIFT and LSHIFT. These are typically three-operand instructions.

Instructions that are intended to operate on a 32-bit register value and use two operands, such as instructions frequently used by a compiler, can use the >>> and >> instructions.

Arithmetic shift, logical shift, and rotate instructions can obtain the shift argument from a register or directly from an immediate value in the instruction. For details about shifter related instructions, see Shifter Instruction Summary.

Two-Operand Shifts

Two-operand shift instructions shift an input register and deposit the result in the same register.

Immediate Shifts

An immediate shift instruction shifts the input bit pattern to the right (down shift) or left (up shift) by a given number of bits. Immediate shift instructions use the data value in the instruction itself to control the amount and direction of the shifting operation.

The following example shows the input value down shifted.

```
R0 contains 0000 B6A3 ;
R0 >>= 0x04 ;
```

results in

```
R0 contains 0000 0B6A ;
```

The following example shows the input value up shifted.

```
R0 contains 0000 B6A3 ;
R0 <<= 0x04 ;
```

results in

```
R0 contains 000B 6A30 ;
```

Register Shifts

Register-based shifts use a register to hold the shift value. The entire 32-bit register is used to derive the shift value, and when the magnitude of the shift is greater than or equal to 32, then the result is either 0 or -1.

The following example shows the input value up shifted.

```
R0 contains 0000 B6A3 ;
R2 contains 0000 0004 ;
R0 <<= R2 ;
```

results in

```
R0 contains 000B 6A30 ;
```

Three-Operand Shifts

Three- operand shifter instructions shift an input register and deposit the result in a destination register.

Immediate Shifts

Immediate shift instructions use the data value in the instruction itself to control the amount and direction of the shifting operation.

The following example shows the input value down shifted.

```
R0 contains 0000 B6A3 ;
R1 = R0 >> 0x04 ;
```

results in

```
R1 contains 0000 0B6A ;
```

The following example shows the input value up shifted.

```
R0.L contains B6A3 ;
R1.H = R0.L << 0x04 ;
```

results in

```
R1.H contains 6A30 ;
```

Register Shifts

Register-based shifts use a register to hold the shift value. When a register is used to hold the shift value (for `ASHIFT`, `LSHIFT` or `ROT`), then the shift value is always found in the low half of a register (`Rn.L`). The bottom six bits of `Rn.L` are masked off and used as the shift value.

The following example shows the input value up shifted.

```
R0 contains 0000 B6A3 ;
R2.L contains 0004 ;
R1 = R0 ASHIFT by R2.L ;
```

results in

```
R1 contains 000B 6A30 ;
```

The following example shows the input value rotated. Assume the Condition Code (`CC`) bit is set to 0. For more information about `CC`, see the discussion of the condition code bit.

```
R0 contains ABCD EF12 ;
R2.L contains 0004 ;
R1 = R0 ROT by R2.L ;
```

results in

```
R1 contains BCDE F125 ;
```

Note the `CC` bit is included in the result, at bit 3.

Bit Test, Set, Clear, Toggle

The shifter provides the method to test, set, clear, and toggle specific bits of a data register. All instructions have two arguments-the source register and the bit field value. The test instruction does not change the source register. The result of the test instruction resides in the CC bit.

The following examples show a variety of operations.

```
BITCLR ( R0, 6 ) ;
BITSET ( R2, 9 ) ;
BITTGL ( R3, 2 ) ;
CC = BITTST ( R3, 0 ) ;
```

When programming, header files (containing #define statements) provide constant definitions for specific bits in memory-mapped registers. It is important to examine the definition techniques used in these header files, because usually the constant definitions do not contain the position of the bit. Rather, header files tend to define bit masks. A constant definition in a header file working with bit masks might be set to 0x20 to describe bit five in a register. The BITPOS command provided by in the Blackfin processor assembler helps when working with bit mask constant definitions and bit manipulation instructions. The following assembly code uses a BITPOS command with a BITTST instruction:

```
#define BITFIVE 0x20
CC = BITTST ( R5, BITPOS ( BITFIVE ) ) ;
```

Note that the BITPOS is calculated at program build time only, not at run time. For detailed information about BITPOS, see the *CrossCore Embedded Studio Assembler and Preprocessor Manual*.

Field Extract and Field Deposit

If the shifter is used, a source field may be deposited anywhere in a 32-bit destination field. The source field may be from 1 bit to 16 bits in length. In addition, a 1- to 16-bit field may be extracted from anywhere within a 32-bit source field.

Two register arguments are used for these functions. One holds the 32-bit destination or 32-bit source. The other holds the extract/deposit value, its length, and its position within the source. For example, if:

- R0 contains 0xAABBCCDD
- R1 contains 0x33331008

where the second byte in R2 (0x10) indicates bit position 16 and the first byte (0x08) indicates the length of the bit field, the zero-extending and sign-extending version return the results:

```
R3 = EXTRACT ( R0 , R1.L ) ( Z ) ;  /* returns 0x000000BB */
R3 = EXTRACT ( R0 , R1.L ) ( X ) ;  /* returns 0xFFFFFFBB */
```

In the deposit instruction uses the upper 16 bits of R1 as data bits. There is a sign-extending version and a non-extending version of the instruction. Zero-extension is not supported:

```
R4 = DEPOSIT ( R0 , R1 ) ( X ) ;  /* returns 0x0033CCDD */
R5 = DEPOSIT ( R0 , R1 )  ;        /* returns 0xAA33CCDD */
```

For details, see the description of the DEPOSIT instruction.

Packing Operation

The shifter also supports a series of packing and unpacking instructions. If:

- R0 contains 0x11223344
- R1 contains 0x55667788

Packing and unpacking operations return:

```
R2 = PACK(R0.L, R0.H); /* returns 0x33441122 */
R3 = PACK(R1.L, R0.H); /* returns 0x77881122 */
R4 = BYTEPACK(R0, R1); /* returns 0x66882244 */
```

The BYTEUNPACK instruction is silently controlled by the *Ix* registers. For example, the instruction

```
(R6, R7) = BYTEUNPACK R1:0;
```

returns:

- R6 = 0x00110022, R7 = 0x00330044, if I0=0
- R6 = 0x00880011, R7 = 0x00220033, if I0=1
- R6 = 0x00770088, R7 = 0x00110022, if I0=2
- R6 = 0x00660077, R7 = 0x00880011, if I0=3

For details, see the descriptions of the BYTEUNPACK instruction and PACK instruction.

Shifter Instruction Summary

The Shifter Instructions and Status table lists the shifter instructions. For more information about assembly language syntax and the effect of shifter instructions on the status bits, see Shifter Instructions and Status.

In the table, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- * Indicates the status bit may be set or cleared, depending on the results of the instruction.
- * 0 Indicates versions of the instruction that send results to Accumulator A0 set or clear AV0.
- * 1 Indicates versions of the instruction that send results to Accumulator A1 set or clear AV1.

- ** Indicates the status bit is cleared.
- *** Indicates CC contains the latest value shifted into it.
- - Indicates no effect.

Table 2-11: Shifter Instructions and Status

Instruction	ASTAT Status Bits						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
BITCLR (Dreg, uimm5) ;	*	*	**	-	-	-	**/-
BITSET (Dreg, uimm5) ;	**	*	**	-	-	-	**/-
BITTGL (Dreg, uimm5) ;	*	*	**	-	-	-	**/-
CC = BITTST (Dreg, uimm5) ;	-	-	-	-	-	*	-
CC = !BITTST (Dreg, uimm5) ;	-	-	-	-	-	*	-
Dreg = DEPOSIT (Dreg, Dreg) ;	*	*	**	-	-	-	**/-
Dreg = EXTRACT (Dreg, Dreg) ;	*	*	**	-	-	-	**/-
BITMUX (Dreg, Dreg, A0) ;	-	-	-	-	-	-	-
Dreg_lo = ONES Dreg ;	-	-	-	-	-	-	-
Dreg = PACK (Dreg_lo_hi, Dreg_lo_hi);	-	-	-	-	-	-	-
Dreg >>>= uimm5 ;	*	*	-	-	-	-	**/-
Dreg >>= uimm5 ;	*	*	-	-	-	-	**/-
Dreg <<= uimm5 ;	*	*	-	-	-	-	**/-
Dreg = Dreg >>> uimm5 ;	*	*	-	-	-	-	**/-
Dreg = Dreg >> uimm5 ;	*	*	-	-	-	-	**/-
Dreg = Dreg << uimm5 ;	*	*	-	-	-	-	*
Dreg = Dreg >>> uimm4 (V) ;	*	*	-	-	-	-	**/-
Dreg = Dreg >> uimm4 (V) ;	*	*	-	-	-	-	**/-
Dreg = Dreg << uimm4 (V) ;	*	*	-	-	-	-	*
A*x* = A*x*>>> uimm5 ;	*	*	-	** 0/-	** 1/-	-	-
A*x* = A*x*>> uimm5 ;	*	*	-	** 0/-	** 1/-	-	-
A*x* = A*x*<< uimm5 ;	*	*	-	* 0	* 1	-	-

Table 2-11: Shifter Instructions and Status (Continued)

Instruction	ASTAT Status Bits						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
Dreg_lo_hi = Dreg_lo_hi >>> uimm4 ;	*	*	-	-	-	-	**/-
Dreg_lo_hi = Dreg_lo_hi >> uimm4 ;	*	*	-	-	-	-	**/-
Dreg_lo_hi = Dreg_lo_hi << uimm4 ;	*	*	-	-	-	-	*
Dreg >>>= Dreg ;	*	*	-	-	-	-	**/-
Dreg >>= Dreg ;	*	*	-	-	-	-	**/-
Dreg <<= Dreg ;	*	*	-	-	-	-	**/-
Dreg = ASHIFT Dreg BY Dreg_lo ;	*	*	-	-	-	-	*
Dreg = LSHIFT Dreg BY Dreg_lo ;	*	*	-	-	-	-	**/-
Dreg = ROT Dreg BY imm6 ;	-	-	-	-	-	***	-
Dreg = ASHIFT Dreg BY Dreg_lo (V) ;	*	*	-	-	-	-	*
Dreg = LSHIFT Dreg BY Dreg_lo (V) ;	*	*	-	-	-	-	**/-
Dreg_lo_hi = ASHIFT Dreg_lo_hi BY Dreg_lo ;	*	*	-	-	-	-	*
Dreg_lo_hi = LSHIFT Dreg_lo_hi BY Dreg_lo ;	*	*	-	-	-	-	**/-
A*x* = A*x* ASHIFT BY Dreg _lo ;	*	*	-	* 0	* 1	-	-
A*x* = A*x* ROT BY imm6 ;	-	-	-	-	-	***	-
Dreg = (Dreg + Dreg) << 1 ;	*	*	*	-	-	-	*
Dreg = (Dreg + Dreg) << 2 ;	*	*	*	-	-	-	*

ADSP-BF70x Compute Units (REGFILE) Register Descriptions

Register File (REGFILE) contains the following registers.

Table 2-12: ADSP-BF70x Compute Units (REGFILE) Register List

Name	Description
Rn	Data Register
A0X	Accumulator 0 Extension Register

Table 2-12: ADSP-BF70x Compute Units (REGFILE) Register List (Continued)

Name	Description
A0	Accumulator 0 Register
A1X	Accumulator 1 Extension Register
A1	Accumulator 1 Register
ASTAT	Arithmetic Status Register

Data Register

The Rn registers (data register file) consists of eight registers, each 32 bits wide. Each register may be viewed as a pair of independent 16-bit registers. Each is denoted as the low half or high half. So, the 32-bit register R0 may be regarded as two independent register halves, R0.L and R0.H.

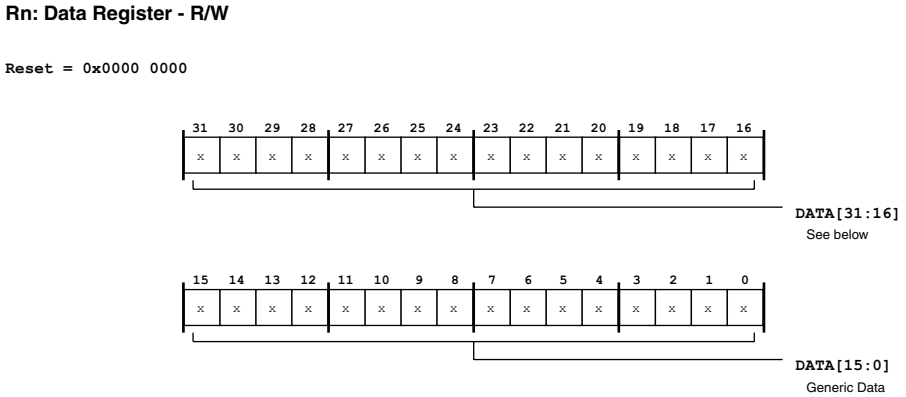


Figure 2-14: Rn Register Diagram

Table 2-13: Rn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Generic Data. The Rn.DATA bit fields--either low half (bits 0:15), high half (bits 16:31), or full (bits 0:31)--hold data for arithmetic operations.

Accumulator 0 Extension Register

The processor has two dedicated, 40-bit accumulator registers, A0 and A1. The parts of each accumulator register may be referred to in code syntax as its 16-bit low half (An.L), its high half (An.H), or its 8-bit extension (An.X). Each accumulator register may also be referred to as a 32-bit register (An.W) consisting of the lower 32 bits, or referred to as a complete 40-bit result register (An).

The accumulator registers may be use together to hold an 80-bit complex result or a 72-bit fixed point result. The combined accumulator register is called A1:0. A 72-bit fixed point result is held in the combined register with least significant bits in A0.W. middle bits in A1.W and most significant bits in A1.X.

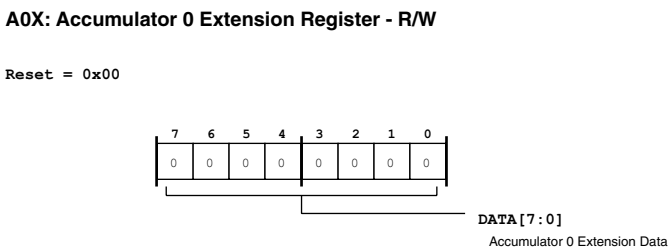


Figure 2-15: A0X Register Diagram

Table 2-14: A0X Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
7:0 (R/W)	DATA	Accumulator 0 Extension Data. The A0X.DATA bits hold 8 bits of accumulator 0 extension data.

Accumulator 0 Register

The processor has two dedicated, 40-bit accumulator registers, A0 and A1. The parts of each accumulator register may be referred to in code syntax as its 16-bit low half (An.L), its high half (An.H), or its 8-bit extension (An.X). Each accumulator register may also be referred to as a 32-bit register (An.W) consisting of the lower 32 bits, or referred to as a complete 40-bit result register (An).

The accumulator registers may be use together to hold an 80-bit complex result or a 72-bit fixed point result. The combined accumulator register is called A1:0. A 72-bit fixed point result is held in the combined register with least significant bits in A0.W. middle bits in A1.W and most significant bits in A1.X.

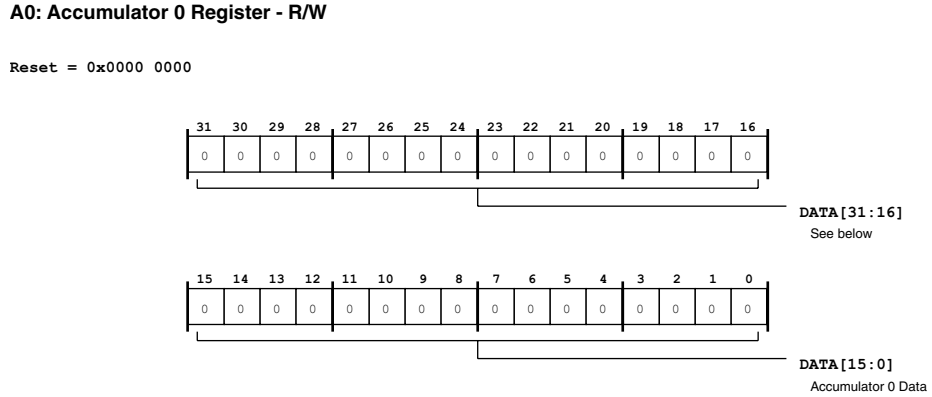


Figure 2-16: A0 Register Diagram

Table 2-15: A0 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Accumulator 0 Data. The A0.DATA bits hold 32 bits of accumulator 0 data.

Accumulator 1 Extension Register

The processor has two dedicated, 40-bit accumulator registers, A0 and A1. The parts of each accumulator register may be referred to in code syntax as its 16-bit low half (An.L), its high half (An.H), or its 8-bit extension (An.X). Each accumulator register may also be referred to as a 32-bit register (An.W) consisting of the lower 32 bits, or referred to as a complete 40-bit result register (An).

The accumulator registers may be use together to hold an 80-bit complex result or a 72-bit fixed point result. The combined accumulator register is called A1:0. A 72-bit fixed point result is held in the combined register with least significant bits in A0.W. middle bits in A1.W and most significant bits in A1.X.

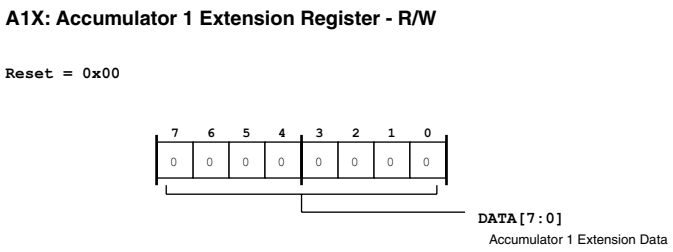


Figure 2-17: A1X Register Diagram

Table 2-16: A1X Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
7:0 (R/W)	DATA	Accumulator 1 Extension Data. The A1X.DATA bits hold 8 bits of accumulator 1 extension data.

Accumulator 1 Register

The processor has two dedicated, 40-bit accumulator registers, A0 and A1. The parts of each accumulator register may be referred to in code syntax as its 16-bit low half (An.L), its high half (An.H), or its 8-bit extension (An.X). Each accumulator register may also be referred to as a 32-bit register (An.W) consisting of the lower 32 bits, or referred to as a complete 40-bit result register (An).

The accumulator registers may be use together to hold an 80-bit complex result or a 72-bit fixed point result. The combined accumulator register is called A1:0. A 72-bit fixed point result is held in the combined register with least significant bits in A0.W. middle bits in A1.W and most significant bits in A1.X.

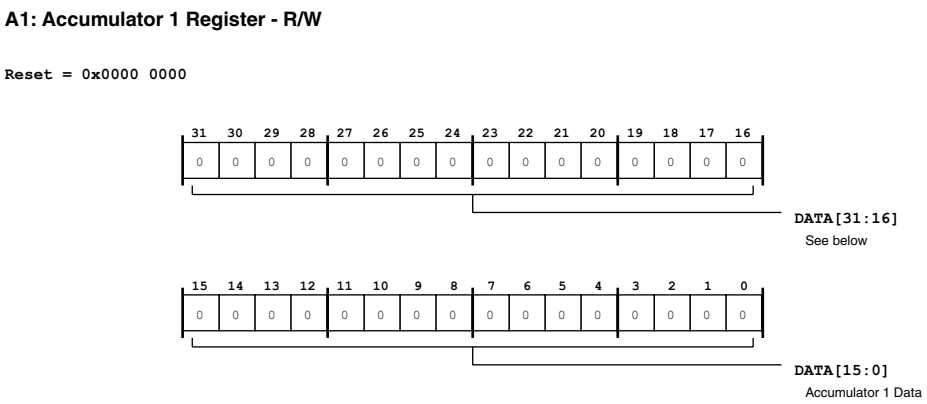


Figure 2-18: A1 Register Diagram

Table 2-17: A1 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Accumulator 1 Data. The A1.DATA bits hold 32 bits of accumulator 1 data.

Arithmetic Status Register

The processor updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation. The ASTAT.AQ bit is updated, indicating the status of the most recent Divs or Divq instruction. The ASTAT.RND_MOD bit does not indicate status, rather this bit selects unbiased or biased rounding for operations supporting rounding.

If execution of an instruction generates status, the instruction's reference page indicates the affected arithmetic status bits.

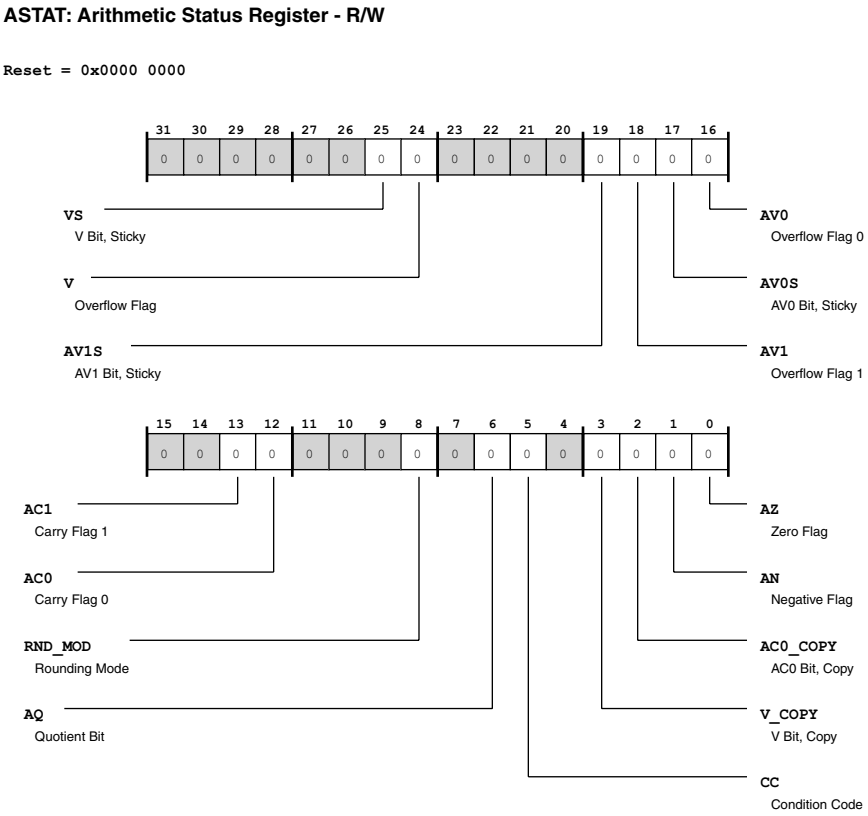


Figure 2-19: ASTAT Register Diagram

Table 2-18: ASTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
25 (R/W)	VS	V Bit, Sticky. The ASTAT.VS bit is set if ASTAT.VS is set; unaffected otherwise.
24 (R/W)	V	Overflow Flag. The ASTAT.V bit is set if the most recent ALU0 or MAC0 operation result overflows; cleared if operation generates no overflow,
19 (R/W)	AV1S	AV1 Bit, Sticky. The ASTAT.AV1S bit is set if ASTAT.AV1 is set; unaffected otherwise.
18 (R/W)	AV1	Overflow Flag 1. The ASTAT.AV1 bit is set if the most recent MAC1 operation result placed in A1 overflows; cleared if result placed in A1 generates no overflow; sticky for MAC.
17 (R/W)	AV0S	AV0 Bit, Sticky. The ASTAT.AV0S bit is set if ASTAT.AV0 is set; unaffected otherwise.
16 (R/W)	AV0	Overflow Flag 0. The ASTAT.AV0 bit is set if the most recent ALU0 or MAC0 operation result placed in A0 overflows; cleared if result placed in A0 generates no overflow; sticky for MAC.

Table 2-18: ASTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
13 (R/W)	AC1	Carry Flag 1. The ASTAT.AC1 bit is set if the most recent ALU1 operation using A1 generates a carry; cleared if operation using A1 generates no carry.	
12 (R/W)	AC0	Carry Flag 0. The ASTAT.AC0 bit is set if the ALU0 operation using A0 generates a carry; cleared if operation using A0 generates no carry.	
8 (R/W)	RND_MOD	Rounding Mode. The ASTAT.RND_MOD bit is not affected by result status. Instead, this bit is used to select the rounding mode for arithmetic instructions that support rounding.	
		0	Unbiased rounding
		1	Biased rounding
6 (R/W)	AQ	Quotient Bit. (Divs and Divq instructions only) The ASTAT.AQ bit equals the dividend MSB exclusive-OR divisor MSB, where the dividend is a 32-bit value and divisor is a 16-bit value.	
5 (R/W)	CC	Condition Code. (TestSet instruction only) The ASTAT.CC bit is set if the addressed value is zero; cleared if the addressed value is non-zero 0x0.	
3 (R/W)	V_COPY	V Bit, Copy. The ASTAT.V_COPY bit is set if ASTAT.V_COPY is set; cleared if ASTAT.V_COPY is cleared.	
2 (R/W)	AC0_COPY	AC0 Bit, Copy. The ASTAT.AC0_COPY bit is set if ASTAT.AC0 is set; cleared if ASTAT.AC0 is cleared.	
1 (R/W)	AN	Negative Flag. The ASTAT.AN bit is set if the most recent ALU0 or shifter operation result is negative; cleared if result is non-negative.	
0 (R/W)	AZ	Zero Flag. The ASTAT.AZ bit is set if the most recent ALU0 or shifter operation result is zero; cleared if result is non-zero.	

3 Operating Modes and States

The processor supports the following three processor modes:

- User mode
- Supervisor mode
- Emulation mode

Emulation and Supervisor modes have unrestricted access to the core resources. User mode has restricted access to certain system resources, thus providing a protected software environment.

User mode is considered the domain of application programs. Supervisor mode and Emulation mode are usually reserved for the kernel code of an operating system. The processor mode is determined by the Supervisor Access (SACC) bit of the SYSCFG register and the Event Controller. When the SACC bit is set, or when servicing an interrupt, a nonmaskable interrupt (NMI), or an exception, the processor is in Supervisor mode. When servicing an emulation event, the processor is in Emulation mode. When not servicing any events and the SACC bit is cleared, the processor is in User mode.

The current processor mode may be identified by interrogating the IPEND memory-mapped register (MMR) and the SACC bit of the SYSCFG register, as shown in the Identifying the **Current Processor Mode** table.

Table 3-1: Identifying the Current Processor Mode

Event	Mode	IPEND	SACC Bit
Interrupt	Supervisor	0x10 but IPEND[0], IPEND[1], IPEND[2], and IPEND[3] = 0.	x
Exception	Supervisor	0x08 The core is processing an exception event if IPEND[0] = 0, IPEND[1] = 0, IPEND[2] = 0, IPEND[3] = 1, and IPEND[15:4] are 0's or 1's.	x
NMI	Supervisor	0x04 The core is processing an NMI event if IPEND[0] = 0, IPEND[1] = 0, IPEND[2] = 1, and IPEND[15:2] are 0's or 1's.	x
Reset	Supervisor	= 0x02 As the reset state is exited, IPEND is set to 0x02, and the reset vector runs in Supervisor mode.	x

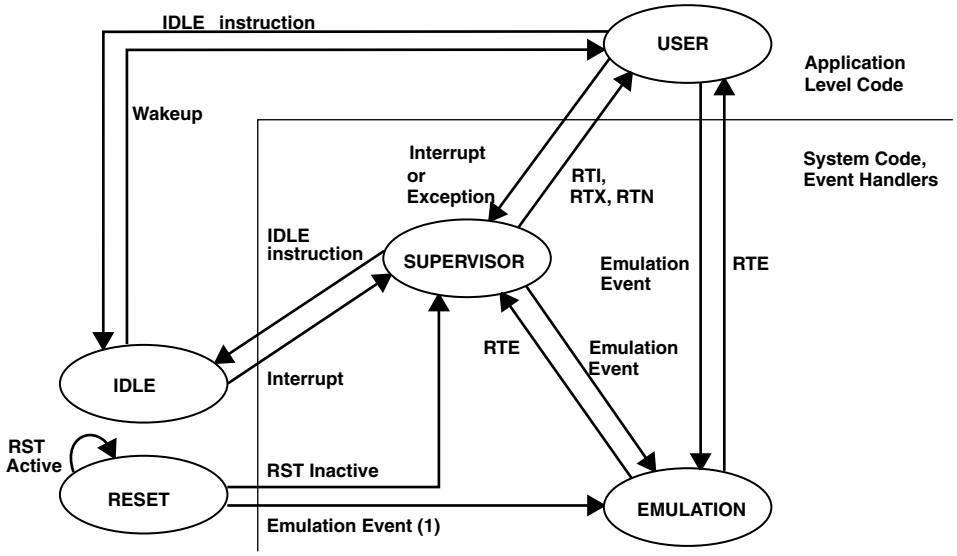
Table 3-1: Identifying the Current Processor Mode (Continued)

Event	Mode	IPEND	SACC Bit
Emulation	Emulator	= 0x01 The processor is in Emulation mode if IPEND[0] = 1, regardless of the state of the remaining bits IPEND[15:1].	x
None	Supervisor	= 0x00	1
None	User	= 0x00	0

In addition, the processor supports the following two non-processing states:

- Idle state
- Reset state

The **Processor Modes and States** figure illustrates the processor modes and states as well as the transition conditions between them when the SACC and STRICT bits in the SYSCFG register are in their default (zero) state.



(1) Normal exit from Reset is to Supervisor mode. However, emulation hardware may have initiated a reset. If so, exit from Reset is to Emulation.

Figure 3-1: Processor Modes and States

User Mode

The processor is in User mode when it is not in Reset or Idle state, when it is not servicing an interrupt, NMI, exception, or emulation event, and when the SACC bit of the SYSCFG register is zero. User mode is used to process application level code that does not require explicit access to system registers. Any attempt

to access restricted system registers causes an exception event. The **Registers Accessible in User Mode** table lists the registers that may be accessed in User mode.

Table 3-2: Registers Accessible in User Mode

Processor Registers	Register Names
Data Registers	R[7:0], A[1:0]
Pointer Registers	P[5:0], SP, FP, I[3:0], M[3:0], L[3:0], B[3:0]
Sequencer and Status Registers	RETS, LC[1:0], LT[1:0], LB[1:0], ASTAT, CYCLES, CYCLES2

Protected Resources and Instructions

System resources consist of a subset of processor registers, all MMRs, and a set of protected instructions.

The system and core MMRs are located in a reserved region of memory which is protected from User mode access. Any attempt to access MMR space in User mode causes an exception. Refer to the specific Blackfin Processor Hardware Reference for the location of the MMR region in your system.

A list of protected instructions appears in the **Protected Instructions** table. Any attempt to issue any of the protected instructions from User mode causes an exception event.

The Strict Supervisor Access (STRICT) bit in the SYSCFG register causes the IDLE instruction to be a protected instruction. When the STRICT bit is set, an attempt to issue the IDLE instruction from User mode causes an exception event.

Table 3-3: Protected Instructions

Instruction	Description
RTI	Return from Interrupt
RTX	Return from Exception
RTN	Return from NMI
CLI	Disable Interrupts
STI	Enable Interrupts
RAISE	Force Interrupt/Reset
STI IDLE	Enable Interrupts and idle
IDLE	Idle Causes an exception only if the STRICT bit in the SYSCFG register is set
RTE	Return from Emulation Always causes an exception if executed outside Emulation mode

Protected Memory

Additional memory locations can be protected from User mode access. A Cacheability Protection Lookaside Buffer (CPLB) entry can be created and enabled. See **Memory Management Unit** in the **Memory** chapter for further information.

Entering User Mode

When coming out of reset, the processor is in Supervisor mode because it is servicing a reset event. To enter User mode from the Reset state, two steps must be performed. First, a return address must be loaded into the RETI register. Second, an RTI must be issued. The SACC bit of the SYSCFG register is zero on reset so this value does not need to be changed to enter User mode on executing the RTI. The following example code shows how to enter User mode upon reset.

Example Code to Enter User Mode Upon Reset

The *Entering User Mode from Reset* example provides code for entering User mode from reset.

```
/* Entering User Mode from Reset */
    P1.L = lo(START) ;   /* Point to start of user code */
    P1.H = hi(START) ;
    RETI = P1 ;
    RTI ;    /* Return from Reset Event */

START :  /* Place user code here */
```

Return Instructions That Invoke User Mode

The **Return Instructions That Can Invoke User Mode** table provides a summary of return instructions that can be used to invoke User mode from various processor event service routines. When these instructions are used in service routines, the value of the return address must be first stored in the appropriate event RETx register. In the case of an interrupt routine, if the service routine is interruptible, the return address is stored on the stack. For this case, the address can be found by popping the value from the stack into RETI. Once RETI has been loaded, the RTI instruction can be issued.

Note a return instruction will only enter User mode if the SACC bit of the SYSCFG register is set to zero. When this bit is set to one the processor remains in Supervisor mode after all event handlers have been exited.

The processor remains in User mode until one of these events occurs:

- An interrupt, NMI, or exception event invokes Supervisor mode.
- An emulation event invokes Emulation mode.
- A reset event invokes the Reset state.

Table 3-4: Return Instructions That Can Invoke User Mode

Current Process Activity	Return Instruction to Use	Execution Resumes at Address in This Register
Interrupt Service Routine	RTI	RETI
Exception Service Routine	RTX	RETX
Nonmaskable Interrupt Service Routine	RTN	RETN
Emulation Service Routine	RTE	RETE

Supervisor Mode

Supervisor mode has full, unrestricted access to all processor system resources, including all emulation resources, unless a CPLB has been configured and enabled. See **Memory Management Unit** in the **Memory** chapter for a further description.

The processor services all interrupt, NMI, and exception events in Supervisor mode. The processor remains in Supervisor mode on return from all event handlers if the SACC bit of the SYSCFG register is set to one.

The stack pointer referenced by the SP register alias and modified by Stack push/pop instructions is the Supervisor Stack Pointer while an event is being serviced (IPEND is non-zero), and the User Stack Pointer when executing at task level (IPEND is zero). This is the case irrespective of whether the processor is running in Supervisor or User mode.

Only Supervisor mode can use the register alias USP, which always references the User Stack Pointer. There is no unique alias for the Supervisor Stack Pointer, so this register can only be referenced within an event handler using the general stack pointer alias, SP.

Normal processing begins in Supervisor mode from the Reset state. The processor transitions from the Reset state to Supervisor mode, servicing the reset event, where it remains until an emulation event or Return instruction occurs to change the mode. Before the Return instruction is issued, the RETI register must be loaded with a valid return address.

Non-OS Environments

For non-OS environments, application code should remain in Supervisor mode so that it can access all core and system resources. On leaving Reset state, the processor initiates operation by servicing the reset event. Emulation is the only event that can preempt this activity. Therefore, lower priority events cannot be processed.

The simplest method of keeping the processor in Supervisor mode and allowing lower priority events to be processed is to set the SACC bit of the SYSCFG register before returning from the reset event with an RTI

instruction. Prior to executing the RTI instruction, RETI must be loaded with the address of the code to be executed after leaving all event handlers.

Earlier Blackfin processors did not have an SACC bit so it was necessary to execute all code in the lowest priority interrupt (IVG15). Events and interrupts are described further in **Events and Interrupts** in the **Program Sequencer** chapter. The interrupt handler for IVG15 is set to the application code starting address, and then the low priority interrupt is forced using the RAISE 15 instruction. The IVG15 interrupt is not serviced until return from the reset event and any pending interrupts with intermediate priorities. So before executing the RTI instruction to return from the reset event RETI is loaded with the address of a loop that executes in User mode until the IVG15 interrupt is serviced.

Example Code for Supervisor Mode Coming Out of Reset

To remain in Supervisor mode when coming out of the Reset state, use code as shown in the **Staying in Supervisor Mode Coming Out of Reset** example.

```
/* Staying in Supervisor Mode Coming Out of Reset */
    R0 = SYSCFG ;
    BITSET (R0, BITP_SYSCFG_SACC) ;
    SYSCFG = R0 ; /* Set SACC bit */
    RETI = START ; /* Set return address to START */
    RTI ; /* Return from Reset Event */

START:
    /* Task level code executes in Supervisor mode */
```

Code written for older Blackfin processors must remain at the lowest interrupt level (IVG15) in order to stay in Supervisor mode as shown in the Staying in Supervisor Mode Coming Out of Reset (Legacy) example.

```
/* Staying in Supervisor Mode Coming Out of Reset (Legacy) */
    P0.L = lo(EVT15) ; /* Point to IVG15 in Event Vector Table */
    P0.H = hi(EVT15) ;
    P1.L = lo(START) ;   /* Point to start of User code */
    P1.H = hi(START) ;
    [P0] = P1 ; /* Place the address of START in IVG15 of EVT */
    P0.L = lo(IMASK) ;
    R0 = [P0] ;
    R1.L = lo(EVT_IVG15) ;
    R0 = R0 | R1 ;
    [P0] = R0 ; /* Set (enable)IVG15 bit in IMASK register */
    RAISE 15 ;   /* Invoke IVG15 interrupt */
    P0.L = lo(WAIT_HERE) ;
    P0.H = hi(WAIT_HERE) ;
    RETI = P0 ;   /* RETI loaded with return address */
    RTI ;   /* Return from Reset Event */

WAIT_HERE :  /* Execute in User mode till IVG15 is serviced */
    JUMP WAIT_HERE ;

START:   /* IVG15 vectors here */
```

```
/* Clears IPEND bit 4 to enable interrupts globally. */
[--SP] = RETI ;
```

Emulation Mode

The processor enters Emulation mode if Emulation mode is enabled and either of these conditions is met:

- An external emulation event occurs.
- The `EMUEXCPT` instruction is issued.

The processor remains in Emulation mode until the emulation service routine executes an `RTE` instruction. If the `SACC` bit of the `SYSCFG` register is zero and no interrupts are pending when the `RTE` instruction executes, the processor switches to User mode. Otherwise, the processor switches to Supervisor mode.

Idle State

Idle state stops all processor activity at the user's discretion, usually to conserve power during lulls in activity. No processing occurs during the Idle state. The Idle state is invoked by an `IDLE` instruction or an `STI IDLE` instruction. The `IDLE` instruction notifies the processor hardware that the Idle state is requested, whereas `STI IDLE` also enables interrupts in a manner that avoids race conditions.

The processor remains in the Idle state until a peripheral or external device, such as a SPORT or the Real-Time Clock (RTC), generates an interrupt that requires servicing.

In Example Code for Transition to Idle State, core interrupts are disabled before the device intended to wake the core from Idle is programmed and the `STI IDLE` instruction is executed. When all the pending processes have completed, the core re-enables interrupts and disables its clocks. The use of the combined `STI IDLE` instruction to enter Idle state and enable interrupts ensures any interrupt will bring the core out of Idle state and terminate the idle instruction, rather than interrupting before the idle instruction has begun execution.

Example Code for Transition to Idle State

To transition to the Idle state, use code shown in the **Transitioning to Idle State** example.

```
/* Transitioning to Idle State */
CLI R0 ;   /* disable interrupts */
/* program wakeup device */
BITSET (R0, BITP_IMASK_IVG11) ;  /* ensure device can interrupt */
STI IDLE R0 ;  /* drain pipeline, enter Idle state, enable interrupts */
```

Reset State

Reset state initializes the core logic. During Reset state, application programs and the operating system do not execute. Clocks are stopped while in Reset state.

The core remains in the Reset state as long as system logic asserts the reset signal. Upon deassertion, the core completes the reset sequence and switches to Supervisor mode with event system priority 1, where it executes code found at an address supplied by the system. Refer to Reset Control Unit (RCU) in the Hardware Reference manual for your processor for details.

The only method of entering Reset state is by receiving a reset signal from the system. The `RAISE 1` instruction will execute the code addressed by `EVT1` at event system priority 1 but does not reset the core.

In both cases an `RTI` instruction will exit the priority 1 event. In neither case is a return address is automatically saved in `RETI` so the register must be explicitly loaded within the event handler.

The Core **State Upon Reset** table summarizes the state of the core upon reset.

Table 3-5: Core State Upon Reset

Item	Description of Reset State
Operating Mode	Supervisor mode in reset event, clocks stopped
Rounding Mode	Unbiased rounding
Cycle Counters	Disabled, zero
DAG Registers (I, L, B, M)	Random values (must be cleared at initialization)
Data and Address Registers	Random values (must be cleared at initialization)
IPEND, IMASK, ILAT	Cleared, interrupts globally disabled with IPEND bit 4
CPLBs	Disabled
L1 Instruction Memory	SRAM (cache disabled)
L1 Data Memory	SRAM (cache disabled)
Cache Validity Bits	Invalid

System Reset and Power Up

For processor specific system reset and power up information, see the corresponding processor hardware reference.

ADSP-BF70x Mode Related (REGFILE) Register Descriptions

Register File (REGFILE) contains the following registers.

Table 3-6: ADSP-BF70x REGFILE Register List

Name	Description
SYSCFG	System Configuration Register

System Configuration Register

The SYSCFG register controls the configuration of the processor. This register is accessible only from the supervisor mode.

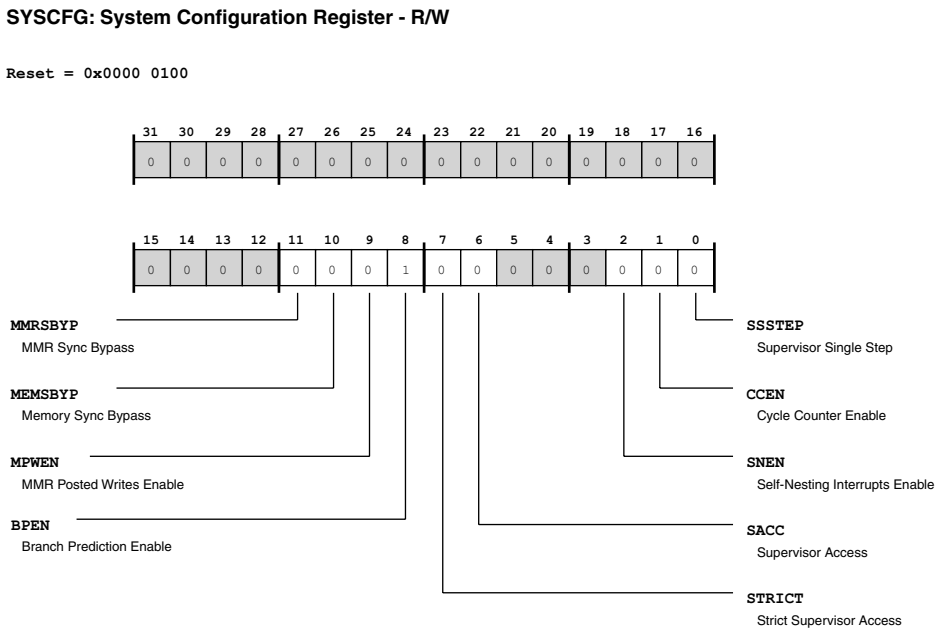


Figure 3-2: SYSCFG Register Diagram

Table 3-7: SYSCFG Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
11 (R/W)	MMRSBYP	MMR Sync Bypass. The SYSCFG.MMRSBYP bit enables bypass mode for clock domain synchronization in memory mapped register interface. Enabling this feature reduces read latency.	
		0	No bypass
		1	Bypass

Table 3-7: SYSCFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
10 (R/W)	MEMSBYP	Memory Sync Bypass. The SYSCFG.MEMSBYP bit enables bypass mode for clock domain synchronization in system memory bus interface. Enabling this feature reduces read latency.	
		0	No bypass
		1	Bypass
9 (R/W)	MPWEN	MMR Posted Writes Enable. The SYSCFG.MPWEN bit enables support for posting consecutive system MMR writes to the system fabric. When disabled, the processor waits for each response before allowing a new write to go into the system. Immediately after changing this mode (either enabling or disabling), an SSYNC instruction must be executed to allow the system to finish any outstanding writes before changing the rules on how writes go out to the system.	
		0	Disable
		1	Enable
8 (R/W)	BPEN	Branch Prediction Enable. The SYSCFG.BPEN bit selects whether the program sequencer uses dynamic or static branch prediction operation. For more information, see the Branch Prediction section of the Program Sequencer chapter.	
		0	Use static prediction
		1	Use dynamic prediction
7 (R/W)	STRICT	Strict Supervisor Access. The SYSCFG.STRICT bit restricts additional resources (beyond those restricted on the ADSP-BF5xx/BF6xx Blackfin processors) to require supervisor mode access. When enabled, accessing any of these additional resources in user mode (for example, executing an IDLE instruction) causes an illegal supervisor access exception. For more information, see the Protected Resources and Instructions section in the Operating Modes and States chapter.	
		0	Disable (normal/previous Blackfin access operation)
		1	Enable (strict supervisor access operation)
6 (R/W)	SACC	Supervisor Access. The SYSCFG.SACC bit selects whether supervisor mode access is permitted when the processor is not servicing any events or whether supervisor mode only is permitted when the processor is servicing an event.	
		0	Disable (access only when servicing an event)
		1	Enable (access whether or not servicing any events)

Table 3-7: SYSCFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
2 (R/W)	SNEN	Self-Nesting Interrupts Enable. The SYSCFG.SNEN bit enables self-nesting interrupt operation. Unlike interrupts during normal operation, interrupts during self-nesting may be interrupted by events at the same priority level. Self-nesting is supported for any interrupt level generated with the RAISE instruction, as well as for core level interrupts.	
		0	Disable (normal interrupt operation)
		1	Enable (self-nesting interrupt operation)
1 (R/W)	CCEN	Cycle Counter Enable. The SYSCFG.CCEN bit enables cycle counter operation. When the cycle counter is enabled, it counts core clock (CCLK) cycles, incrementing with each cycle. All cycles are counted (including wait states) in both user mode and supervisor modes. The cycle counter stops counting while the processor is in emulator mode. The cycle counter is 64 bits wide, and the count is stored in the CYCLES and CYCLES2 registers. The least significant 32 bits are stored in CYCLES.	
		0	Disable cycle counter
		1	Enable cycle counter
0 (R/W)	SSSTEP	Supervisor Single Step. The SYSCFG.SSSTEP bit enables single step operation, in which a supervisor exception occurs after the processor executes each instruction. This bit only applies to executing instructions in user mode or to processing interrupts in supervisor mode. The SYSCFG.SSSTEP bit is ignored if the core is processing an exception or higher-priority event. If precise exception timing is required, a CSYNC instruction must be used after setting this bit.	
		0	Disable (normal operation)
		1	Enable (single step operation)

4 Program Sequencer

This chapter describes the Blackfin processor program sequencing and interrupt processing modules. For information about instructions that control program flow, see the program flow control instruction reference pages. For information about instructions that control interrupt processing, see the external event management chapter. Discussion of derivative-specific interrupt sources can be found in the hardware reference for the specific part.

Introduction

In the processor, the program sequencer controls program flow, constantly providing the address of the next instruction to be executed by other parts of the processor. Program flow in the chip is mostly linear, with the processor executing program instructions sequentially.

The linear flow varies occasionally when the program uses nonsequential program structures, such as those illustrated in the program flow variations figure. Nonsequential structures direct the processor to execute an instruction that is not at the next sequential address. These structures include:

- **Loops.** One sequence of instructions executes several times with zero overhead.
- **Subroutines.** The processor temporarily interrupts sequential flow to execute instructions from another part of memory.
- **Jumps.** Program flow transfers permanently to another part of memory.
- **Interrupts and Exceptions**. A runtime event or instruction triggers the execution of a subroutine.
- **Idle**. An instruction causes the processor to stop operating and hold its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

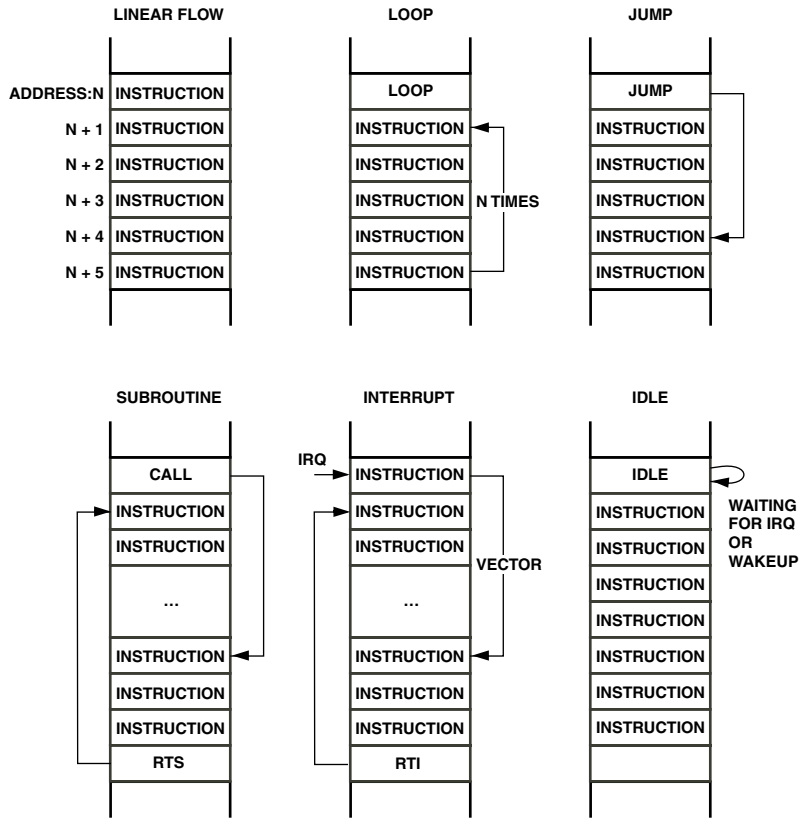


Figure 4-1: Program Flow Variations

The sequencer manages execution of these program structures by selecting the address of the next instruction to execute.

The dynamic branch predictor runs ahead of the sequencer anticipating the next instruction address that the sequencer will select so the instruction pipeline can be kept busy when the changes of control flow are predictable.

The fetched address enters the instruction pipeline, ending with the program counter (`PC`). The pipeline contains the 32-bit addresses of the instructions currently being fetched, decoded, and executed. The `PC` couples with the `RETx` registers, which store return addresses. All addresses generated by the sequencer are 32-bit memory instruction addresses.

To manage events, the event controller handles interrupt and event processing, determines whether an interrupt is masked, and generates the appropriate event vector address.

In addition to providing data addresses, the data address generators (DAGs) can provide instruction addresses for the sequencer's indirect branches.

The sequencer evaluates conditional instructions and loop termination conditions. The loop registers support nested loops. The memory-mapped registers (MMRs) store information used to implement interrupt service routines.

The block diagram shows the core Program Sequencer module and how it interconnects with the Core Event Controller (CEC).

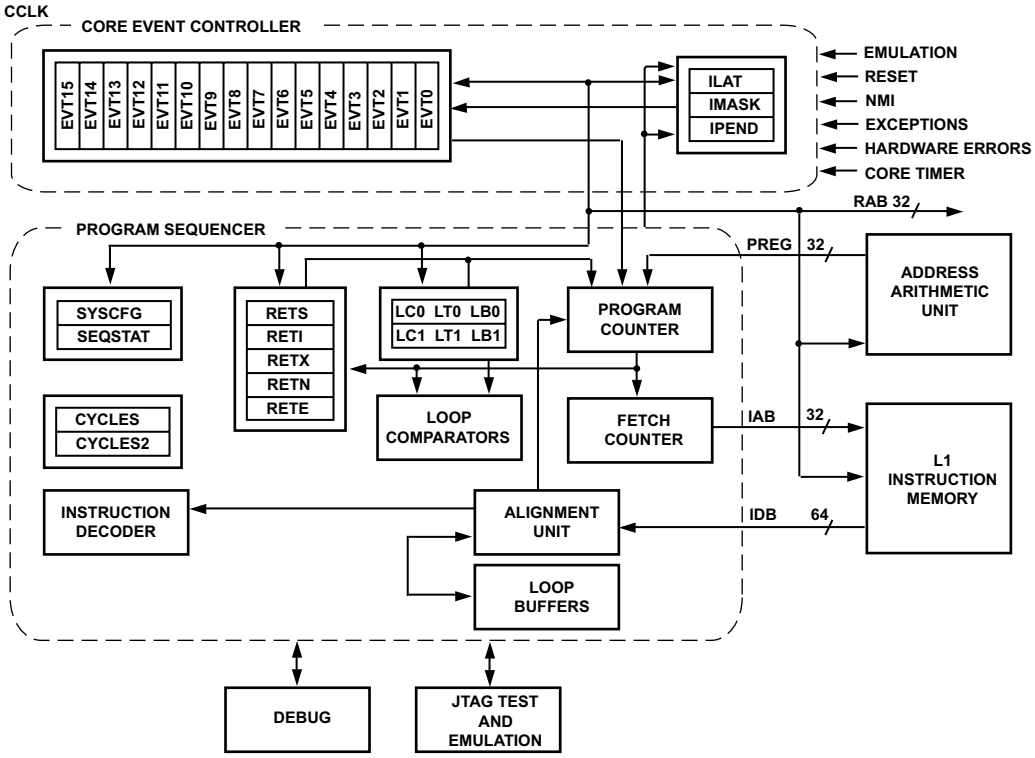


Figure 4-2: Program Sequencing and Interrupt Processing Block Diagram

Sequencer Related Registers

The sequencer registers table lists the non-memory-mapped registers within the processor that are related to the sequencer. Except for the `PC` register, all sequencer-related registers are directly readable and writable by move instructions, for example:

```
SYSCFG = R0 ;

P0 = RETI ;
```

Manually pushing or popping registers to or from the stack is done using the explicit instructions:

```
[--SP] = Rn ; /* for push */

Rn = [SP++] ; /* for pop */
```

Similarly, all non-memory-mapped sequencer registers can be pushed and popped to or from the system stack:

```
[--SP] = CYCLES ;
```

```
SYSCFG = [SP++] ;
```

However, load/store operations and immediate loads are not supported.

Table 4-1: Non-memory-mapped Sequencer Registers

Register Name	Description
SEQSTAT	Sequencer Status register.
RETX RETN RETI RETE RETS	Return Address registers: Exception Return NMI Return Interrupt Return Emulation Return Subroutine Return
LC0, LC1 LT0, LT1 LB0, LB1	Zero-Overhead Loop registers: Loop Counters Loop Tops Loop Bottoms
FP, SP	Frame Pointer and Stack Pointer
SYSCFG	System Configuration register.
CYCLES, CYCLES2	Cycle Counters.
PC	Program Counter. The PC is an embedded register. It is not directly accessible with program instructions.

In addition to these central sequencer registers, there is a set of memory-mapped registers that interact closely with the program sequencer. For information about the interrupt control registers, see Events and Interrupts. Although the registers of the Core Event Controller are memory-mapped, they still connect to the same 32-bit Register Access Bus (RAB) and perform in the same way. Registers of the System Event Controller resides in the SYSCLK domain. For debug and test registers, see the processor hardware reference.

Instruction Pipeline

The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. If no conditions require otherwise, the processor executes instructions from memory in sequential order by incrementing the look- ahead address.

The processor has a ten-stage instruction pipeline, shown in the table.

Table 4-2: Stages of Instruction Pipeline

Pipeline Stage	Description
Instruction Fetch 1 (IF1)	Issue instruction address to IAB bus, start compare tag of instruction cache

Table 4-2: Stages of Instruction Pipeline (Continued)

Pipeline Stage	Description
Instruction Fetch 2 (IF2)	Wait for instruction data
Instruction Fetch 3 (IF3)	Read from IDB bus and align instruction
Instruction Decode (DEC)	Decode instructions
Address Calculation (AC)	Calculation of data addresses and branch target address
Data Fetch 1 (DF1)	Issue data address to DA0 and DA1 bus, start compare tag of data cache
Data Fetch 2 (DF2)	Read register files
Execute 1 (EX1)	Read data from LD0 and LD1 bus, start multiply and video instructions
Execute 2 (EX2)	Execute/Complete instructions (shift, add, logic, etc.)
Write Back (WB)	Writes back to register files, SD bus, and pointer updates (also referred to as the "commit" stage)

The figure shows a diagram of the pipeline.

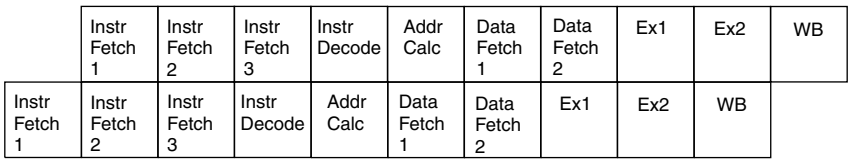


Figure 4-3: Processor Pipeline

The instruction fetch and branch logic generates 32-bit fetch addresses for the Instruction Memory Unit. If a dynamic branch predictor is present and enabled the fetch address may be generated based upon a dynamic prediction of decisions made by the sequencer later in the pipeline. The Instruction Alignment Unit returns instructions and their width information at the end of the IF3 stage.

For each instruction type (16-, 32-, or 64-bit), the Instruction Alignment Unit ensures that the alignment buffers have enough valid instructions to be able to provide an instruction every cycle. Since the instructions can be 16, 32, or 64 bits wide, the Instruction Alignment Unit may not need to fetch an instruction from the cache every cycle. For example, for a series of 16-bit instructions, the Instruction Alignment Unit gets an instruction from the Instruction Memory Unit once in four cycles. The alignment logic requests the next instruction address based on the status of the alignment buffers. The sequencer responds by generating the next fetch address in the next cycle, provided there is no change of flow.

The sequencer holds the fetch address until it receives a request from the alignment logic or until a mis-predict is detected. If no change of flow is predicted, the sequencer increments the previous fetch address by 8 (the next 8 bytes). A mis-predict occurs when the dynamic branch predictor is disabled and a change of flow occurs, such as a branch or an interrupt, or the predictor is enabled but failed to correctly anticipate the next fetch address. When a mis-predict occurs data in the Instruction Alignment Unit is invalidated. The sequencer decodes and distributes instruction data to the appropriate locations such as the register file and data memory.

Data register file reads occur in the DF2 pipeline stage (for operands).

Data register file writes occur in the WB stage (for stores). The multipliers and the video units are active in the EX1 stage, and the ALUs and shifter are active in the EX2 stage. The accumulators are written at the end of the EX2 stage.

The program sequencer also controls stalling and invalidating the instructions in the pipeline. Multi-cycle instruction stalls occur between the IF3 and DEC stages. DAG and sequencer stalls occur between the DEC and AC stages. Computation and register file stalls occur between the DF2 and EX1 stages. Data memory stalls occur between the EX1 and EX2 stages.

NOTE: The sequencer ensures that the pipeline is fully interlocked and that all the data hazards are hidden from the programmer.

Multi-cycle instructions behave as multiple single-cycle instructions being issued from the decoder over several clock cycles. For example, the Push Multiple or Pop Multiple instruction can push or pop from 1 to 14 `Dregs` and/or `Pregs`, and the instruction remains in the decode stage for a number of clock cycles equal to the number of registers being accessed.

Multi-issue instructions are 64 bits in length and consist of one 32-bit instruction and two 16-bit instructions. All three instructions execute in the same amount of time as the slowest of the three.

Any nonsequential program flow can potentially decrease the processor's instruction throughput. Nonsequential program operations include:

- Jumps
- Subroutine calls and returns
- Interrupts and returns
- Loops

Branches

One type of nonsequential program flow that the sequencer supports is branching. A branch occurs when a `JUMP` or `CALL` instruction begins execution at a new location other than the next sequential address. For descriptions of how to use the `JUMP` and `CALL` instructions, see the program flow control instruction reference pages. Briefly:

- A `JUMP` or a `CALL` instruction transfers program flow to another memory location. The difference between a `JUMP` and a `CALL` is that a `CALL` automatically loads the return address into the `RETS` register. The return address is the next sequential address after the `CALL` instruction. This load makes the address available for the `CALL` instruction's matching return instruction, allowing easy return from the subroutine.
- A return instruction causes the sequencer to fetch the instruction at the return address, which is stored in the `RETS` register (for subroutine returns). The types of return instructions include: return from subroutine (`RTS`), return from interrupt (`RTI`), return from exception (`RTX`), return from emulation

(RTE), and return from nonmaskable interrupt (RTN). Each return type has its own register for holding the return address.

- A JUMP instruction can be conditional, depending on the status of the CC bit of the ASTAT register. These instructions are immediate and may not be delayed. The program sequencer can evaluate the CC status bit to decide whether to execute a branch. If no condition is specified, the branch is always taken.
- Conditional JUMP instructions use static branch prediction to reduce the branch latency caused by the length of the pipeline, when dynamic branch prediction is not enabled.

Branches can be direct or indirect. A direct branch address is determined solely by the instruction word (for example, JUMP 0x30), while an indirect branch gets its address from the contents of a DAG register (for example, JUMP(P3)).

All types of JUMPs and CALLs can be PC-relative. The indirect JUMP and CALL can be absolute or PC-relative.

Direct Jumps (Short, Long and Extra Long)

The sequencer supports three lengths of direct jump, short, long and extra long. In all cases the target of the branch may be a PC-relative address from the location of the instruction, plus an offset. The PC-relative offset for the short jump is a 13-bit immediate value that must be a multiple of two (bit 0 must be a 0). The 13-bit value gives an effective dynamic range of -4096 to +4094 bytes.

The PC-relative offset for the long jump is a 25-bit immediate value that must also be a multiple of two (bit 0 must be a 0). The 25-bit value gives an effective dynamic range of -16,777,216 to +16,777,214 bytes.

The PC-relative offset for the extra long jump is a 32-bit immediate value that must also be a multiple of two (bit 0 must be 0). The 32-bit value gives an effective dynamic range of the entire address space of the processor.

At the time of writing the program the offset of the destination is usually unknown. The development tools will evaluate the offset and select the appropriate jump length when the JUMP 0xnnnnnnnn instruction is used.

The exact form of jump length can be specified by using a JUMP.S 0xnnnn instruction if a 13-bit offset is required, a JUMP.L 0xnnnnnnn instruction if a 25-bit offset is required or a JUMP.XL 0xnnnnnnnn instruction if a 32-bit is required. For an absolute jump target address JUMP.A 0xnnnnnnnn should be used. Upon disassembly, a JUMP instruction is replaced by the appropriate JUMP.S, JUMP.L or JUMP.XL instruction.

Rather than hard coding jump target addresses, use symbolic addresses in assembly source files. Symbolic addresses are called labels and are marked by a trailing colon. See the *Visual DSP++ Assembler and Preprocessor Manual* for details.

```
JUMP mylabel ;
/* skip any code placed here */
```

```
mylabel:
/* continue to fetch and execute instruction here */
```

Direct Call (Long and Extra Long)

A call instruction is a branch instruction that copies the address of the instruction which would have executed next (had the call instruction not executed) into the RETSregister. The development tools will evaluate the offset and select the appropriate PC-relative call instruction when the CALLinstruction syntax is used.

The long direct call instruction has a 25-bit, PC-relative offset that must be a multiple of two (bit 0 must be a 0). The 25-bit value gives an effective dynamic range of –16,777,216 to +16,777,214 bytes. A long direct call can be specified explicitly by using a CALL.Linstruction.

The extra long direct call instruction has a 32-bit PC-relative offset that must be a multiple of two (bit 0 must be a 0). The 32-bit value gives an effective dynamic range of the entire address space of the processor. An extra long PC-relative direct call can be specified explicitly by using a CALL.XLinstruction. An absolute direct call is obtained by using the CALL.Ainstruction.

Indirect Jump and Call (Absolute)

The indirect JUMPand CALLinstructions get their destination absolute address (branch target) from a data address generator (DAG) pointer register. For the CALLinstruction, the RETSregister is loaded with the address of the instruction which would have executed next in the absence of the CALL instruction.

For example:

```
JUMP (P3) ;
CALL (P0) ;
```

On Blackfin+ processors a pointer register may be loaded directly with a 32-bit value.

```
P4 = mytarget;
JUMP (P4);
mytarget:
/* continue here */
```

Earlier Blackfin processors could only load a 16-bit value to a pointer register half. To load a pointer register with a 32-bit symbolic target value in two steps you may use one of the following syntax styles. The syntax may differ in various assembly tools sets. Modern style:

```
P4.H = hi(mytarget);
P4.L = lo(mytarget);
JUMP (P4) ;
mytarget:
/* continue here */
```

Legacy style:

```
P4.H = mytarget;
P4.L = mytarget;
JUMP (P4) ;
mytarget:
/* continue here */
```

Indirect Jump and Call (PC-Relative)

The PC-relative indirect JUMP and CALL instructions use the contents of a pointer register as an offset to the branch target. For the CALL instruction, the RETS register is loaded with the address of the instruction which would have executed next (had the CALL instruction not executed).

For example:

```
JUMP (PC + P3) ;
CALL (PC + P0) ;
```

Subroutines

Subroutines are code sequences that are invoked by a `CALL` instruction. Assuming the stack pointer `SP` has been initialized properly, a typical scenario could look like the following:

```
/* parent function */
R0 = 0x1234 (Z); /* pass a parameter */
CALL myfunction;
/* continue here after the call */
[P0] = R0; /* save return value */
JUMP somewhereelse;
myfunction: /* subroutine label */
[--SP] = (R7:7, P5:5); /* multiple push instruction */
P5.H = hi(myregister); /* P5 used locally */
P5.L = lo(myregister);
R7 = [P5]; /* R7 used locally */
R0 = R0 + R7; /* R0 used for parameter passing */
(R7:7, P5:5) = [SP++]; /* multiple pop instruction */
RTS; /* return from subroutine */
myfunction.end: /* closing subroutine label */
```

Due to the syntax of the multiple-push, multiple-pop instructions, often the upper data registers and pointer registers are used for local purposes, while lower registers pass the parameters. See the address arithmetic unit chapter for more details on stack management. The `CALL` instruction not only redirects the program flow to the *myfunction* routine, it also writes the return address into the `RETS` register. The `RETS` register holds the address where program execution resumes after the `RTS` instruction executes. In the example this is the location that holds the `[P0]=R0;` instruction. The return address is not passed to any stack in the background. Rather, the `RETS` register functions as single-entry hardware stack. This scheme enables "leaf functions" (subroutines that do not contain further `CALL` instructions) to execute with less

possible overhead, as no bus transfers need to be performed. If a subroutine calls other functions, it must temporarily save the content of the RETS register explicitly. Most likely this is performed by stack operations as shown below.

```
/* parent function */
CALL function_a;
/* continue here after the call */
JUMP somewhereelse;
function_a: /* subroutine label */
[--SP] = (R7:7, P5:5); /* optional multiple push instruction */
[--SP] = RETS; /* save RETS onto stack */
CALL function_b; /* call further subroutines */
CALL function_c;
RETS = [SP++]; /* restore RETS */
(R7:7, P5:5) = [SP++]; /* optional multiple pop instruction */
RTS; /* return from subroutine */
function_a.end: /* closing subroutine label */
function_b:
/* do something */
RTS;
function_b.end:
function_c:
/* do something else */
RTS;
function_c.end:
```

Stack Variables and Parameter Passing

Many subroutines require input arguments from the calling function and need to return their results. Often, this is accomplished by project-wide conventions, that certain core registers are used for passing arguments, where others return the result. It is also recommended that assembly programs meet the conventions used by the C/C++ compiler. See the *CrossCore Embedded Studio C/C++ Compiler and Library Manual* for details.

Extensive arguments are typically passed over the stack rather than by registers. The following example passes and returns two 32-bit arguments:

```
_parent:
...
R0 = 1;
R1 = 3;
[--SP] = R0;
[--SP] = R1;
CALL _sub;

R1 = [SP++];   /* R1 = 4 */
R0 = [SP++];   /* R0 = 2 */
...
_parent.end:

_sub:
```

```
[--SP] = FP;   /* save frame pointer */
FP = SP;   /* new frame */
[--SP] = (R7:5);   /* multiple push */

R6 = [FP+4];   /* R6 = 3 */
R7 = [FP+8];   /* R7 = 1 */
R5 = R6 + R7;   /* calculate anything */
R6 = R6 - R7;

[FP+4] = R5;   /* R5 = 4 */
[FP+8] = R6;   /* R6 = 2 */

(R7:5) = [SP++];   /* multiple pop */
FP = [SP++];   /* restore frame pointer */
RTS;
_sub.end:
```

Since the stack pointer SP is modified inside the subroutine for local stack operations, the frame pointer FP is used to save the original state of SP. Because the 32-bit frame pointer itself must be pushed onto the stack first, the FP is four bytes off the original SP value.

The Blackfin instruction set features a pair of instructions that provides cleaner and more efficient functionality than the above example: the LINK and UNLINK instructions. These multi-cycle instructions perform multiple operations that can be best explained by the equivalent code sequences shown in the table.

Table 4-3: Link and Unlink Code Sequence Equivalents

LINK n;	UNLINK;
--SP] = RETS; ">[--SP] = RETS; --SP] = FP; ">[--SP] = FP; FP = SP; SP += -n;	SP = FP; FP = [SP++]; RETS = [SP++];

The following subroutine does the same job as the one above, but it also saves the RETS register to enable nested subroutine calls. Therefore, the value stored to FP is 8 bytes off the original SP value. Since no local frame is required, the LINK instruction gets the parameter "0".

```
_sub2:
LINK 0;
[--SP] = (R7:5);

R6 = [FP+8];   /* R6 = 3 */
R7 = [FP+12];   /* R7 = 1 */

R5 = R6 + R7;
R6 = R6 - R7;

[FP+8] = R5;   /* R5 = 4 */
[FP+12] = R6;   /* R6 = 2 */
```

```
(R7:5) = [SP++];
UNLINK;
RTS;
_sub2.end:
```

If subroutines require local, private, and temporary variables beyond the capabilities of core registers, it is a good idea to place these variables on the stack as well. The `LINK` instruction takes a parameter that specifies the size of the stack memory required for this local purpose. The following example provides two local 32-bit variables and initializes them to zero when the routine is entered.

```
_sub3:
LINK 8;
[--SP] = (R7:0, P5:0);
R7 = 0 (Z);
[FP-4] = R7;
[FP-8] = R7;
...
(R7:0, P5:0) = [SP++];
UNLINK;
RTS;
_sub3.end:
```

For more information, see the `LINK` and `UNLINK` instruction reference pages.

Conditional Processing

The Blackfin processors support conditional processing through conditional jump and move instructions. Conditional processing is described in the following sections:

- Condition Code Status Bit
- Conditional Branches
- Branch Prediction
- Speculative Instruction Fetches
- Conditional Register Move

Condition Code Status Bit

The processor supports a Condition Code (CC) status bit, which is used to resolve the direction of a branch. This status bit may be accessed nine ways:

- A conditional branch is resolved by the value in CC.
- A Data register value may be copied into CC, and the value in CC may be copied to a Data register. For example,

```
R0 = CC; /* R0 becomes either 0 or 1 */

CC = R1;
```

- The BITTST instruction accesses the CC status bit. For example,

```
CC = BITTST (R0, 31) ;

/* CC set to value of bit 31 in R0 */
```

- A status bit may be copied into CC, and the value in CC may be copied to a status bit. For example,

```
CC = AV0;
```

- The CC status bit may be set to the result of a Pointer register comparison. For example,

```
CC = P0 < P1 ;
```

- The CC status bit may be set to the result of a Data register comparison. For example,

```
CC = R5  == R7;
```

- Some shifter instructions (rotate or BXOR) use CC as a portion of the shift operand/result. For example,

```
R0 = rot R1 by R1.L ;
```

- Store-exclusive and SYNCEXCLinstructions can set and clear CC bit. For example,

```
CC = ([P5] = R0) (P5) ;
```

- Test and set instructions can set and clear the CC bit. For example,

```
TESTSET (P5) ;
```

These nine ways of accessing the CC bit are used to control program flow. The branch is explicitly separated from the instruction that sets the arithmetic status bits. A single bit resides in the instruction encoding that specifies the interpretation for the value of CC. The interpretation is to "branch on true" or "branch on false. "

The comparison operations have the form CC = expr where *expr* involves a pair of registers of the same type (for example, Data registers or Pointer registers, or a single register and a small immediate constant). The small immediate constant is a 3-bit (-4 through 3) signed number for signed comparisons and a 3-bit (0 through 7) unsigned number for unsigned comparisons.

The sense of CC is determined by equal (==), less than (<), and less than or equal to (<=). There are also bit test operations that test whether a bit in a 32-bit data register is set.

Conditional Branches

The sequencer supports conditional branches. Conditional branches are JUMP instructions whose execution branches or continues linearly, depending on the value of the CC bit. The target of the branch is a PC-relative address from the location of the instruction, plus an offset. The PC-relative offset is an 11-bit immediate value that must be a multiple of two (bit 0 must be a 0). This gives an effective dynamic range of -1024 to +1022 bytes.

For example, the following instruction tests the CC status bit and, if it is positive, jumps to a location identified by the label dest_address:

```
IF CC JUMP dest_address ;
```

Similarly, a branch can also be taken when the CC bit is not set. Then, use the syntax:

```
IF !CC JUMP other_addr ;
```

NOTE: Take care when conditional branches are followed by load operations. For more information, see the Load/Store instruction reference pages.

Branch Prediction

Branches can be accelerated if the processor predicts the target of an upcoming branch instruction before it has been committed and starts fetching instructions from the target sooner reducing the number of instructions that need to be aborted. Prediction can be performed dynamically based upon the location and direction of the branches the processor has previously executed, or prediction may be static based upon information supplied by the programmer.

Some Blackfin processor implementations support dynamic branch prediction. Software can check for this capability by testing the BPREDbit of the FEATUREOMMR. Older Blackfin processor implementations which do not have a FEATUREOMMR do not support dynamic branch prediction either.

Dynamic branch prediction is enabled by setting the BPENbit of the SYSCFGregister. Once enabled, the branch latency of all branches, whether taken or not taken, is significantly reduced. The latency is never longer than the latency of the branch were it statically mispredicted.

The dynamic predictor comes up in a state in which it can immediately start executing. There may be some delay before it can improve the latency of branches, but software does not need to do anything other than set BPENto configure it. The static prediction bit in conditional branch instructions is used to initialize the dynamic prediction. Prediction is validated before the instruction at the branch address is committed, so self-modifying code and code overlays will work as expected with the predictor enabled.

Additional control registers are provided for test, verification and tuning of the dynamic branch predictor. Software should not need to use these registers to achieve reasonable performance in the general case. For more information on the dynamic branch predictor see Dynamic Branch Prediction

Statistically dynamic branch prediction is far more successful than static prediction and can lead to significant improvements in program performance without code modification. Its use is recommended. However the predictor consumes power. So, disabling it and relying only on static prediction might be considered for low power applications.

When dynamic branch prediction is disabled or not available, the sequencer supports static branch prediction to accelerate execution of conditional branches. These branches are executed based on the state of the CC bit.

In the EX2 stage, the sequencer compares the actual CC bit value to the predicted value. If the value was mispredicted, the branch is corrected, and the correct address is available for the WB stage of the pipeline.

The branch latency for statically predicted conditional branches is as follows.

- If static prediction was "not to take branch," and branch was actually not taken: 0 CCLK cycles.
- If static prediction was "not to take branch," and branch was actually taken: 8 CCLK cycles.
- If static prediction was "to take branch," and branch was actually taken: 4 CCLK cycles.
- If static prediction was "to take branch," and branch was actually not taken: 8 CCLK cycles.

For all unconditional branches, the branch target address computed in the AC stage of the pipeline is sent to the Instruction Fetch Address bus at the beginning of the DF1 stage. All statically predicted unconditional branches have a latency of 4 CCLK cycles.

Consider the example in the table.

Table 4-4: Branch Prediction

Instruction	Description
If CC JUMP dest (bp)	This instruction tests the CC status bit, and if it is set, jumps to a location, identified by the label, dest. When dynamic branch prediction is disabled or not available (bp) indicates the branch is predicted taken. If the CC status bit is set, the branch is correctly predicted and the branch latency is reduced. Otherwise, the branch is incorrectly predicted and the branch latency increases. When dynamic branch prediction is enabled (bp) is simply used to initialize the predictor. Branches that are always taken, or always not taken will tend to be correctly predicted and have a greatly reduced branch latency.

Dynamic Branch Prediction

Dynamic branch prediction is performed by a unit in the Blackfin processor known as the Branch Predictor (BP). The default configuration of the BP should be suitable for most programs and once the BP is enabled further configuration should not be necessary. However this section provides a detailed description of the BF70x BP so programmers can gain an insight into how the performance of their programs is affected, and possibly tune the predictor to improve performance of their particular program. The details of BP behavior and registers are specific to this implementation of the Blackfin processor.

Operation of the BP is controlled by BPEN bit in the System Configuration (SYSCFG) register.

Branch Predictor Overview

To improve branch execution performance, the BF70x BP learns the type, source address, and target address of most static and conditional branches. For conditional branches, it will also learn and update a 4 value prediction code that indicates which direction the branch is most likely to take based on recent execution history. This information is stored in a 16K bit RAM referred to as the BP Table. Each time an instruction memory fetch is made, the memory address is used to access the BP Table to see if the BP has learned a branch associated with that address. If there is a static branch or a conditional branch that is predicted taken in the table, the BP provides the type instruction and target address to the fetch block which immediately fetches the target. Branch predictions made by the BP result in a savings of 5 cycles for static branches and 8 cycles for conditional branches when compared with non-predicted branches.

The BP performs two kinds of access to the BP Table.

A fetch access is triggered every time the sequencer fetches an instruction from memory. The BP uses the fetch address to read from the table and check for possible branch hits. If one is found, the branch data is evaluated and a prediction is sent to the instruction fetch block which then fetches the target address provided by the BP.

Management accesses add new entries to the table or modify existing entries. Management accesses are only performed in cycles when there are no instruction fetches. This ensures that BP predictions have a higher priority even at the expense of additional delay in adding or updating branch entries to the table.

The BP has two 32-bit memory mapped registers (MMRs), BP_CFG and BP_STAT, to control and monitor its operation.

BP RAM

The BF70x fetch unit operates on instruction data that is 64 bits wide with addresses on 64 bit boundaries. Each 64 bit segment is referred to as a Line. One Line is fetched for each instruction fetch. BP is designed to predict 2 possible branches for each Line.

The BP Table RAM can be viewed as 2 way set associative. Each row will hold the data for 2 branches or 1 Line. The first branch learned for a Line will be associated with Set 0 and the second branch with Set 1. Additional branches will be added by alternating and overwriting Set 0 or Set 1. The data in each set can be divided into two 32 bit parts, the TAG and the TARG. The TAG section contains the branch source information and the TARG contains the branch destination or target address.

The BP uses 64 bits for each table entry in Set 1 or Set 0. The LRU bit points to the oldest branch table entry that was accessed for each Line. This data is used to determine which set to overwrite when a new branch is to be learned. It is written to both the TAG0 and TAG1 portions of the row whenever either Set0 or Set1 is written. The other bit assignments for the branch TAG and TARG portions of table entries can be seen below.

Table 4-5: BP Table Entry Structure

Field	Enum	Name	Description
TAG[31:10]			The 21 most significant bits of the branch source address.
TAG[9]		SPAN	Indicates when an instruction spanned 2 Lines. Obsolete.
TAG[8:6]		TYPE	
	000	BRCC	Conditional Jump
	001	JMP	Unconditional Jump
	010	RTS	Return from Subroutine
	011	Not used	Not used
	100	CALL16	Not Used
	101	CALL32	Short Call
	110	CALL64	Long Call
	111	CALL128	Long Call multi Instruction
TAG[5:4]		PREDICTION	Prediction Strength
	11	Strongly Taken	
	10	Weakly Taken	
	01	Weakly Not Taken	
	00	Strongly Not Taken	
TAG[3]		VALID BIT	Can be set to 0 to remove entry from prediction process.
TAG[2:1]		SOF	Byte offset of the branch in the Line. Source Address[2:1].
TAG[0]		LRU Bit	
TARG[31:0]			The 32 bits of the branch target address

Configuring The Branch Predictor

Branch Predictor support is controlled by the `BPEN` bit in the System Configuration (`SYSCFG`) register. If the BP is disabled, it will finish any fetch or management table access that is in progress but will not begin a new access.

The following table shows the branch instructions supported by the BP. The BP type code which is stored with the branch TAG data is shown following each instruction type.

Table 4-6: Branch Instructions Supported by BP

JUMP pcrel13	BP Type JUMP	001
JUMP.L pcrel24	BP Type JUMP	001
JUMP Imm32(opt pcrel)	BP Type JUMP	001
IF CC0 JUMP pcrel10	BP Type BRCC	000
IF CC0 JUMP pcrel10 (bp)	BP Type BRCC	000

Table 4-6: Branch Instructions Supported by BP (Continued)

IF !CC0 JUMP pcrel10	BP Type BRCC	000
IF !CC0 JUMP pcrel10 (bp)	BP Type BRCC	000
CALL pcrel24	BP Type CALL32	101
CALL Imm32(opt pcrel)	BP Type CALL64	110
RTS	BP Type RTS	010

The following table shows the branch instructions NOT supported by the BP.

Table 4-7: Branch Instructions NOT Supported by BP

JUMP (Px)
JUMP (PC + Px)
CALL (Px)
CALL (PC + Px)
RTE
RTI
RTX
RTN
LSETUP (Note: Hardware loops are zero latency without the BP).

The BP provides enable bits for each of the branch types that it supports. These bits can be found in `BP_CFG` MMR. When the enable bits are set to 1, the BP will execute requests to learn branches of a specific type and add them to the Table. When set to 0, new branches for the specified type will not be learned but branches of this type that are already in the table will continue to be predicted and updated.

Prediction for all branch types should be enabled for best average performance across the most general range of programs. Prediction for individual branch types may be disabled to tune the BP operation to a specific program. Experiments to find a better specific configuration, if it exists, should begin by measuring program performance with all enable bits set.

BP Store Buffers

The BP receives information from the Sequencer pipe control logic which tells it when to learn and update data about branches that it is executing. This information comes to the BP at several places in the pipe and on different phases of the clock. This requires the BP to align and store data in order to enter it into the Table RAM in the same access. The strategy of executing management table operations between instruction fetches, which are not easily predictable, also requires the BP to buffer data before it is entered into the table.

To meet these requirements, the BP has 2 data store buffers. These buffers store the data coming from the Sequencer that is used to load the TAG and TARG portions of the Table entry. They also store information such as whether to learn or update as well as data required for updating prediction states.

Each store buffer is managed by a 3 value state machine. The states can be idle, waiting for additional data, or full. The store buffer enters the full state once it has all of the data that it needs to complete each type of table access operation. When the buffer is full it generates a request to the table state control machine to move its data to the table. It waits in the full state until the table state control machine accepts its data and begins writing it to the table. The store buffer machine can then move to idle, waiting, or full with a new buffer full of data.

The BP has additional store buffer logic which controls the next buffer to load and the order that buffer requests are fed to the table state control machine.

BP Table Control

The table control state machine manages 4 types of table accesses which can be requested by the store buffers. The types are learn, update, instruction mispredict, and address mispredict. A learn access creates a new entry in the table and writes TAG and TARG data to either Set 0 or Set 1 of the table row indicated by bits 9:3 of the branch source address found in the TAG. An update access changes the prediction values in the TAG fields of Set 0 or Set 1 based on the branch source address and information provided by the Sequencer when the predicted branch is executed and an update is requested.

The use of self modifying code and code overlays can cause the BP Table to contain outdated branch entries which will cause the BP to send incorrect predictions to the Sequencer. Two additional types of table accesses correct this problem. An instruction mispredict access occurs when the type of a prediction does not match the type expected by the Sequencer. When this occurs the Sequencer requests an instruction mispredict access and provides the offending source address. The table control state machine then executes an instruction mispredict access which sets the Valid bit to 0 in the TAG field of the appropriate entry in the table. This prevents further predictions from this entry. An address mispredict access occurs when the type of prediction matches what the Sequencer expects but the target address does not. When this occurs, the Sequencer requests an address mispredict access and provides the offending source and correct target address. The table control state machine then executes an address mispredict access which updates the TARG field of the appropriate entry in the table with the correct target address. This insures that future predictions to this source address will point to the correct target address.

The LRU bit is toggled and updated for each of the 4 types of table accesses.

To support the 4 types of table accesses, the table control state machine has 7 states. The states are idle, check, process, learn, update, instruction mispredict, and address mispredict. The idle state occurs when there are no access requests from the store buffers. The check state moves the data from a requesting store buffer to a local table control buffer. It also executes a RAM read using bits 9:3 of the source address from the local buffer for the RAM address. The state machine may remain in the check state for several cycles until a non-fetch cycle is available for the read to execute in. The process state is entered once the check state read completes. Data from the check state read is used during the process state to determine if the branch in the local buffer was found in the table, calculate new prediction values if the access is an update, set the next valid bit to 0 if the access is an instruction mispredict, and set the next value for the LRU bit. The process state always requires only 1 cycle and instruction fetches and predictions can be performed while it is executing. Following the process state, the state machine will move into one of the remaining 4 states, learn, update, instruction mispredict, or address mispredict depending on the request type contained in the local buffer. Each of these types will execute a RAM write using data obtained from the

local buffer or calculated during the process cycle. The 4 different states enable different RAM write control lines depending on the type of data which needs to be written for each access type. The state machine may remain in the one of these 4 write states for several cycles until a non-fetch cycle is available for the write to execute in. On completion of the write, the state machine will return to idle or can move to the check state and begin processing the next store buffer request immediately.

The local control buffer managed by the table control state machine stores the same data as a store buffer. The requesting store buffer is freed as soon as the check state is entered so three table access requests may be in flight at any given time.

The table control state machine waits in check state and write state until a non-fetch cycle is available. To ensure this wait is not indefinite the number of sequential fetch cycles are counted and if they exceed a threshold the Sequencer is requested to hold off an instruction fetch for one cycle. The STMOUTVAL field of the BP_CFG register holds the threshold value, and the STMOUTCNTR field of the BP_STAT register holds the current value of the counter.

All types of table access perform a table read when in check state. If the access is a learn request a matching entry is not expected to be found in the table. If a matching entry is found the DFL bit in the BP_STAT register is set and the new data is not written to the table. This condition can occur when multiple learn requests are issued by the sequencer as a result of a delayed entry of the first request into branch table. If the access is an update, instruction mispredict or address mispredict and the entry is not found, the NFL bit in the BP_STAT register is set and no data is written to the table. This can occur when a table entry is overwritten with a new branch entry just after it is used to make a prediction. Both these conditions can occur during the normal operation of the BP. The DFL and NFL bits are sticky and are reset by writing 1 to the CLRDFL and CLRNFL bits of the BP_CFG register.

Table Initialization

The Table RAM contains several pieces of data that are used for control purposes which must be in a known state before the Table can be used for predictions. These data types include the LRU bits, the valid bits, and the prediction value bits. For this reason, the entire table is initialized whenever the core is reset.

The initialization process for the Table is triggered whenever reset is deasserted. All of the entries in the Table are written with 0s, one row at a time. This means that the initialization process requires approximately 132 cycles. During the initialization, all other types of accesses to the table are blocked so no predictions, learning, or register Table accesses will be performed. This means that there is no BP operation for the first 132 cycles after reset is deasserted.

During the initialization, the Table is unavailable but the Store Buffers are allowed to operate. This means that learning and update requests will be loaded to the Store Buffers. They will remain in the buffers until the Table is done initializing and is ready to accept Store Buffer requests. The result of this is that the last 2 BP operations requested by the Sequencer during the initialization period will be loaded to the Table once it has completed its initialization. The operations requested by the Sequencer before the last 2 requests will be lost.

The BP also provides the capability to re-initialize the Table while the core is not in reset. This feature may be useful in situations where code overlays are being swapped. In this situation, it may be more cycle efficient to remove stale branches from the table before learning branches associated with a new code block.

This prevents correcting the stale branches one at a time through instruction or address mispredict operations. The usefulness of this approach will need to be evaluated on a case by case basis.

To reinitialize the BP Table, set bit 0 of the BP_CFG register to 1 and write the register. This bit is self-clearing. Writing this BP Table Clear bit will trigger the same initialization process that occurs when reset is deasserted. The BP will begin predicting and learning with a clean Table after 132 cycles.

Sequencer BP Requests

The Sequencer produces 4 types of requests which are loaded to the Store Buffers and then used by the BP to modify the Table. These types are learn, update, instruction mispredict, and address mispredict. The Sequencer provides the data for these requests at 2 different points during pipe execution. The timing information for these requests is provided below.

The BP receives the earliest requests in pipe stage "E". These are referred to as mid-pipe requests. The targets for these branches are received in pipe stage "F/G". The second group of requests occurs in pipe stage "J". These are referred to as late-pipe requests. The targets for these branches are received at the same time, in pipe stage "J".

Mid-pipe requests include learns for most static branches including JUMPS, CALLS, and RTS. Dynamic or conditional branches (BRCCs) which are predicted taken (BP argument) are also learned at mid-pipe. BRCCS which are not predicted taken (no BP argument) are not learned mid-pipe. The Instruction Mispredict request is also asserted at mid pipe.

Update requests for all branch types occur in the late pipe. BRCCs which are not predicted taken (no BP argument) and are taken (mispredicted) are learned in the late pipe. BRCCs which are predicted taken (BP argument) and are not taken (mispredicted) were learned at mid pipe and are not re-learned in the late pipe. The first prediction for these branches will be incorrect but will be updated to the proper value in the update following the first prediction. Address mispredict requests are asserted by the Sequencer in pipe stage "H". The new target value for address mispredicts isn't asserted until pipe stage "J" so the BP treats Address Mispredicts as late pipe requests.

The Sequencer performs update requests to modify the prediction values in the table based on whether the BP dynamic branch predictions were correct or not. Updates are also used to monitor the number of predictions made and to modify the LRU bit in the BP Table entry so that older entries are overwritten first. By default, the Sequencer will generate an update for every BP prediction it receives including predictions for static branches such as unconditional Jumps, CALL's and RTS's. The BP has two alternative modes in which the table is updated less frequently. These are selected by setting the SKUPD or SKUPDLRU bits in the BP_CFG register. Only one of these bits should be set at a time.

Skip Update Mode is enabled when the SKUPD bit of the BP_CFG register is set. This mode causes updates to be skipped if the prediction code for a predicted branch is strongly taken or strongly not taken. Since the prediction code is set to strongly taken for all static branches, updates will be eliminated for all static branches. Updates for BRCCs will be eliminated if the prediction code is strongly taken or strongly not taken and the prediction was not mispredicted. Mispredicted BRCCs generate an update request regardless of the update mode so the prediction value can be updated.

Skip Update LRU Mode is enabled when the SKUPDLRU bit of the BP_CFG register is set. This mode causes updates to be skipped if the prediction code for a predicted branch is strongly taken or strongly not taken and the predicted branch is not the oldest (LRU) in the table for a specific Line. The additional LRU qualification has the result of keeping the newest accessed branch in the table longer. This should increase the frequency of predictions for branches located near the current PC.

The Skip Update and Skip Update LRU modes sacrifice the accuracy of prediction to reduce the frequency of updates. An excessive number of updates might increase the delay before branches are actually learned and impact the frequency of instruction fetches so this can be a net benefit, but in the general case the default setting is expected to perform best.

BP Store Buffer State Machine

The basic Store Buffer state machine operation is as follows. When a mid pipe request is assigned to one of the store buffers in pipe stage "E", its state moves from Idle to Wait. The wait is required because we have to wait until pipe stage "F/G" to receive the target address. Once the target is received, the BP has all of the required data and the state moves from Wait to Full. A request is then sent to the Table Control state machine to transfer the requested operation and data to the Table Control local buffer so it can be used to modify the Table. After the request is accepted and the data is moved, the Store Buffer state machine can return to Idle. When a late pipe request is assigned to a store buffer, the state machine moves directly from Idle to Full because all of the required data is available in the same cycle. When the state moves to Full, the interaction with the Table Control State machine is the same as in the mid pipe request case. The store buffer data is moved and the Store Buffer state machine can return to idle.

There are several exceptions to this basic operation due to sequencer operation or to reduce the number of cycles that it takes for a request to move through the store buffers and be incorporated into the Table.

The first exception occurs when a late-pipe request is sent to the BP before or at the same time as the target for a mid-pipe request. The state machine would have moved from Idle to Wait when the mid-pipe request was received. If a late-pipe request is received at this point, it is given a higher priority because late-pipe requests will typically cause a change of flow (COF) which means the mid-pipe request will be killed and its target never fetched or learned. When the late-pipe request is received, the Store Buffer loads the data as in a normal late-pipe request and the state machine moves to Full. The transfer of the late request then moves to the Table in the normal fashion. The mid-pipe data is overwritten and the mid-pipe request is dropped.

The next exception to note is that mid-pipe transactions can be lost if another branch which is not supported by the BP causes a COF to occur while the state machine is in Wait. Interrupts are a good example of this. When a mid-pipe operation starts, the state machine moves to Wait. If an interrupt COF occurs before the mid-pipe target is fetched and sent to the BP, the correct target will never be sent to the BP to complete the mid-pipe learn. In this case the non-learned COF causes the mid-pipe request to be dropped and the state returns to Idle. This scenario does not occur for late-pipe requests because all of the data is presented in the same cycle and a late pipe learn will not occur in the same cycle as any other non-supported branch COF. This means that the BP has everything it needs to complete the late-pipe request so it executes without a problem.

A third exception to note is that if requests from the sequencer come close together, the store buffer state machine does not have to return to Idle before starting a new request. It is possible for the state machine to move from Full to Wait or from Full to Full again with the appropriate new mid or late pipe requests. If the state machine is Full when an new request is started and the request to move the current data to the Table Control local buffer has not been accepted, the current data will be overwritten. The number of times that data is overwritten is an important measure of Store Buffer performance so overwrites are included as one of the BP event monitoring parameters.

The last exception to note is that mid-pipe instruction mispredicts are treated differently than other mid-pipe requests. Instruction mispredicts occur when a branch is predicted which is no longer valid. It is important to correct this problem quickly to prevent the branch from being predicted a second time and incurring the unneeded fetch stalls again. To speed up these requests, the instruction mispredict request is treated as a late-pipe request even though it occurs at mid-pipe. When the request is received, it causes the Store Buffer state machine to move from Idle to Full. This is possible since the target address normally associated with mid-pipe requests is not needed to execute the request. After the state machine is Full, the request moves to the Table Control buffer in the usual fashion. The special handling of these requests allows them to move through the store buffers in 1 cycle.

The final aspect of Store Buffer operation which needs to be discussed is the interactions between the Store Buffers. Store buffers interact in 2 ways. These are the order that Store Buffer requests are sent to the Table Control buffer for execution and the order in which Sequencer requests are loaded to the buffers.

The order that requests are sent to the Table Control buffer for execution is straightforward. The most recently loaded or newest store buffer request is sent to the Table Control buffer first. If a request has already been asserted but has not been accepted by the Table Control state machine, it will be deasserted and the newer request will be asserted. The older request may be reasserted when the newer request is accepted by the Table Control State machine. This policy has the result of getting branches close to the current PC into the Table as quickly as possible. This increases the probability of these branches being predicted in tight loops and thus will increase the efficiency of the BP.

The loading of Store Buffers with Sequencer requests is controlled by several policies. The first is that the first buffer loaded after reset is always Store Buffer 0. The second is that when a Store Buffer is written and its state goes to Full, the other buffer is selected as the next Store Buffer to be written. This policy results in loads being alternated between the Store buffers. The contents of the oldest Store Buffer will be overwritten if all of the requests have not been executed to the Table when a new request is received. This behavior is seen most frequently and occurs when requests are not closely spaced together.

The third policy is that Store Buffers which have just had their data accepted by the Table Control state machine will now be considered empty and will be loaded next. This policy is in effect for only 1 cycle per request. It is the cycle when the Table Control state machine informs the Store Buffers that it has completed a request and can accept data for the next request. This cycle is typically the cycle after a Table write which is the last cycle of the completed Table request. Once this single cycle is completed, control of buffer loading reverts to the policy of alternating buffers. This third policy will reduce the amount of older Store Buffer data which is overwritten and dropped thus increasing BP efficiency. It occurs when branches are spaced close together and a new request has been allowed to move to the Table ahead of a request that is waiting. Several new requests may be loaded and executed to the table before an older request has an opportunity to be loaded. It should be noted that because of this third policy, the order and timing of Store

Buffers loads is now dependent on the exact cycle when requests are completed by the Table Control state machine. The execution of requests by the Table Control state machine is also a strong function of when instructions are fetched. Because the timing of instruction fetches is highly variable, the net effect is that timing and loading of Store Buffer requests is also a function of the timing of instruction fetches and is highly variable. This third policy can also cause older requests to be delayed a long time before actually being entered into the table.

There is one other case where Store Buffer loading is affected. This is the first exception case described above which occurs when a late-pipe request is sent to the BP before or at the same time as the target for a mid-pipe request. As previously noted, the late-pipe request will override the mid-pipe request. This will change the data in the Store Buffer, move the buffer state to Full, and cause this buffer to be marked as the newest buffer. This will also change the order of buffer requests executed to the Table even though it did not change the next buffer which was selected to be loaded.

It should be noted that the policy of loading the most recent requests to the Table first is also important in achieving quick table loads when handling instruction mispredict requests. When an instruction mispredict request is received, it is the newest request so it is sent to the table immediately. This allows the BP to guarantee a 1 cycle throughput through the store buffers for this type of request.

The order and timing for loading Sequencer requests to the Store buffers will affect when branch entries enter the Table and thus BP predictions and program performance. As has been discussed, the order, timing, and potential overwriting of requests to the Store Buffers are highly variable. To provide some visibility and debug capability for this process, the Full State bits for each of the Store Buffer state machines have been brought out to the BP Status register. These bits are updated every cycle and can be used to observe Store Buffer operation.

BP Predictions

The BP Table is checked for branch hits each time an instruction fetch is executed. Checks are triggered within the BP in pipe stage "A" by the signal fetch_a1f which comes from the Sequencer Fetch block. Predictions are available approximately 1 ½ cycles after the instruction fetch is started. This short path time is a major factor in determining BP performance and presented the most difficult challenge in the BP design.

The first step in generating a prediction is to determine if there are branch entries found in the Line that is being fetched. The BP supports predictions for 2 branch entries per Line. This requires that the BP also determine where each branch instruction is located within the Line. Since each Line is 64 bits or 8 bytes and the minimum instruction size is 2 bytes, the maximum number of instructions per Line is 4. This means that 2 bits are required to locate the start of an instruction or instruction offset (SOF) within a Line and that the SOF corresponds to bits 2:1 of the instruction address.

When an instruction fetch is evaluated for a prediction, the BP compares the 22 bit TAG's and the 2 bit SOFs in both Sets 0 and 1 of the appropriate Table Row with the corresponding bits in the fetch address. If the TAGS match, the SOF of the fetch address is less than or equal to the SOF of the entry, and the Valid bit is true, then the entry is a branch hit. If the TAG's match, and the Valid bit is set, but the SOF of the fetch address is greater than the SOF of the entry then there is no hit.

If there are no branch hits for a given fetch address then there will be no predictions for that instruction fetch.

If there is a hit in either Set0 or Set 1, but not both, then the data from the set which produced the hit will be used to generate the prediction. In this case, a multiplexer on the BP outputs will be switched to send the required SOF, Type, and Target Address data from the Set which hit to the Sequencer and Hoppers. The final prediction evaluation, however, depends on the prediction value found in the Table for the branch entry. If the prediction value is strongly or weakly taken, the hit creates a Taken Prediction and the Sequencer is signaled to fetch the Target value provided by the BP. If the prediction value is strongly or weakly not taken, the hit creates a Not Taken Prediction and the Sequencer is not signaled to fetch the target. Not Taken Predictions are useful to monitor BP performance but are not sent to the Sequencer so no COF or BP Updates occur.

A more complex prediction process happens when 2 hits are found in the Table for a given fetch address. When this occurs, the prediction is determined using the SOFs and prediction values for each branch entry. The cases which are possible are discussed below. In each case, the TAGs are assumed to match the fetch address and the Valid bits are set for both Table Entries. Branch A may be in Set 0 or 1 with Branch B located in the other set.

1. Fetch SOF <= Branch A SOF < Branch B SOF - The prediction value for Branch A is Strongly or Weakly Taken and the value for Branch B is a don't care. In this case, a Taken prediction will be issued for Branch A. Branch A has the lower offset so it will be first in the pipe. Since it is predicted taken, it will be fetched to avoid the fetch overhead caused by its COF. The presence and prediction value for Branch B are not factors because Branch B comes after Branch A in the pipe and will be killed when Branch A is executed.

2. Fetch SOF <= Branch A SOF < Branch B SOF - The prediction value for Branch A is Strongly or Weakly Not Taken and the value for Branch B is Strongly or Weakly Taken. In this case, a Taken Prediction will be issued for Branch B. Branch A has the lower offset so it will be first in the pipe but it is not taken so a prediction and a fetch are not necessary. Branch B will be executed because Branch A is not taken. Since Branch B will be executed and is predicted taken, issuing the prediction for it will be useful in avoiding the fetch overhead caused by its COF.

3. Branch A SOF < Fetch SOF <= Branch B SOF - The prediction value for Branch A is a don't care and the value for Branch B is Strongly or Weakly Taken In this case, a Taken Prediction will be issued for Branch B. This will happen when a branch is executed which has a target address offset which is between that of Branch A and B. Branch A has a lower offset but it will not be in the pipe and will never be executed so there is no need to predict it. Branch B is predicted taken and will be executed so a prediction for Branch B will be useful in avoiding the fetch overhead caused by its COF.

4. Fetch SOF <= Branch A SOF < Branch B SOF - The prediction value for Branches A and B are Strongly or Weakly Not Taken. In this case, a Not Taken Prediction will be issued for Branch A because it has the lower SOF. A prediction will not be created for Branch B because it will be fetched when Branch A is fetched and, since it is not taken, a fetch and prediction will not be required to load its target.

5. Branch A SOF < Fetch SOF <= Branch B SOF - The prediction value for Branches A and B are Strongly or Weakly Not Taken. In this case, a Not Taken Prediction will be issued for Branch B. The prediction will not be issued for Branch A because the SOF of the target address is greater than the SOF for Branch

A. The SOF of the target address is less than or equal to the SOF for Branch B so Branch B is a valid hit and a Not Taken Prediction is issued for it.

6. Branch A SOF < Branch B SOF < Fetch SOF - The prediction value for Branches A and B are don't cares. In this situation, no predictions are issued because no branches exist in the Line above Branch B.

Once the correct set and prediction type has been identified as described in the above cases, the BP prediction multiplexer is switched to the appropriate set and the prediction signals and data are sent to the Sequencer and Hoppers as described for the single hit case above.

As presented in the BP RAM Design Section, there are 4 values for the Prediction Code which can assigned to each Branch Table entry. These are Strongly Taken, Weakly Taken, Weakly Not Taken, and Strongly Not Taken. When a static branch such as a Call or RTS is learned, its prediction value is assigned as Strongly Taken. This value does not change when updates are performed for these entries. When a conditional branch (BRCC) is learned, its initial prediction value is assigned as Weakly Taken. This value is recomputed each time the branch is predicted and an Update is executed for the entry.

Predictions which are predicted taken and are taken (not mispredicted) cause the value to move 1 prediction value towards Strongly Taken. Once the prediction code is set to Strongly Taken, it will remain in this state through taken Updates until a mispredict Update occurs. This will cause the state to move to Weakly Taken.

Predictions which are predicted taken and are not taken (mispredicted) cause the value to move 1 prediction value towards Strongly Not Taken. Once the prediction code is set to Strongly Not Taken, it will remain in this state until a mispredict Update occurs. This will cause the state to move to Weakly Not Taken.

Branch predictions are not directly affected by memory stalls. The BP will check for a branch hit each time an instruction is fetched. If a hit is found and a prediction made, the BP will hold the prediction signals and data on its outputs until the next instruction fetch. If the new fetch produces a hit, the prediction signals and data will be changed to reflect the new prediction. If the new fetch does not produce a hit, the control signals are deasserted to indicate there is no prediction. The data, however, is a don't care and may or may not change depending on the previous and current data fetched from the Table. When an instruction fetch occurs and is stalled, the BP will check for a prediction in the first cycle of the fetch because fetch_a1f and the instruction address are both asserted in this cycle. The results of this check are held on the BP outputs through the stall until the next fetch, as usual. This allows the BP data to be used at the end of the stall when it is required by the Sequencer and Hoppers. The only effect that memory stalls have on the BP is that they change the timing of when fetches occur. As discussed, BP efficiency is affected by the number and timing of instruction fetches so it is possible to see a change in BP operation when there is a high density of memory stalls.

In general, BP predictions will occur whenever there is an instruction fetch. There is one exception to this rule. When the BP makes a prediction, the target address of the branch is fetched in the next cycle. It is possible that the branch fetched may span into the next address so the second address must also be fetched. This additional fetch is referred to as the trailer. The policy of determining whether a predicted branch required a trailer fetch or not lead to a difficult timing situation so a simpler policy regarding trailer fetches was implemented. The new policy is that when a Taken Prediction is made, the target address and the trailer address will always be fetched and loaded to the Hoppers. Because the trailer address will be used

only for branches which span and the branch is taken, branches which occur later in the trailer will never be executed. If they will never be executed there is no need to predict them so the BP does not check and make predictions for trailer addresses which are fetched.

The BP is responsible for learning branches and detecting when the branches that it has learned are being fetched but it is not always responsible for providing the target address of the branch. For RTS branches, the target address comes from the Call Return Stack block. In each case, the BP sends signals to these blocks to notify them that a RTS has been fetched and these blocks provide the target from their internal storage. The target address for RTS instructions will come from one of the 8 locations in the Call Return Stack or the RETS register.

The final note regarding predictions is that data found in the BP Table when a prediction is made is sent to the ILAT so that it can manage its Instruction Hoppers. This information includes the number of branches found for the Line, the types and offsets for each branch, the Set where the predicted branch is found, whether the predicted branch is taken or not, and the Skip Update mode which is passed on to the Sequencer. The ILAT uses this information to determine which Hoppers are no longer needed and may be reused for new instruction data. This increases Hopper efficiency by reducing the number of Hopper stalls required and thus increases overall program efficiency.

Speculative Instruction Fetches

The pipeline architecture requires the program sequencer to speculatively fetch instructions that may have to be discarded. A useful example for this operation is the sequence:

```
CC = P0 == 0;
if CC jump skip;
csync;
call (P0);
skip:
```

Even without the shown `CSYNC` instruction, the sequence is fully functional. The call would not be taken if `P0` was zero. However, without having the `CSYNC` instruction there, the instruction fetch from `P0` would still happen. Since address 0x0000 0000 resides in SDRAM memory space, the sequence would trigger an instruction fetch from SDRAM that could be unwanted. If the SDRAM controller has not been initialized properly, the conditional instruction fetch would even trigger a hardware error. Thus, the `CSYNC` instruction is recommended. See the load/store instruction reference pages for details on related data load topics.

Conditional Register Move

Register moves can be performed depending on whether the value of the `CC` status bit is true or false (1 or 0). In some cases, using this instruction instead of a branch eliminates the cycles lost because of the branch. These conditional moves can be done between any data registers or pointer registers (including `SP` and `FP`).

Example code:

```
IF CC R0 = P0 ;
IF !CC P1 = P2 ;:
```

Hardware Loops

The sequencer supports a mechanism of zero-overhead looping. The sequencer contains two loop units, each containing three registers. Each loop unit has a Loop Top register (LT0, LT1), a Loop Bottom register (LB0, LB1), and a Loop Count register (LC0, LC1).

Zero-overhead loops are most conveniently written with the LOOP/LOOP_END construct. The loop start is marked with a LOOP, LOOPZ or LOOPLEZ instruction and the end of the loop marked with a LOOP_END pseudo-instruction.The following code example shows a loop that contains two instructions and iterates 32 times.

```
LOOP LC0 = 32 ;
  R5 = R0 + R1(ns) || R2 = [P2++] || R3 = [I1++] ;
  R5 = R5 + R2 ;
LOOP_END ;
```

Loops that begin with the LOOPinstruction decrement and test the counter at the end of the loop and exit if it is zero. At least one iteration of the loop is always executed.

Additionally the LOOPZand LOOPLEZinstructions test the counter before the first iteration of the loop and only execute the first iteration if the counter is initially within range. The LOOPZinstruction jumps to the instruction after the LOOP_ENDwhen the counter is initially zero. The LOOPLEZinstruction jumps to the instruction after the LOOP_ENDwhen the counter is initially less than or equal to zero. When the counter is initially in range then the LOOPZand LOOPLEZoperate in the same way as the LOOPinstruction. See the LSETUP, LOOP instruction reference page for operation details.

The following code shows two loops with an unknown trip count. In the first loop the LOOPinstruction is used so at least one iteration is executed. In the second loop the LOOPLEZinstruction is used so the exactly the count iterations are executed.

```
P5 = [P4];
LOOP LC1 = P5;
  /* loop body executed at least once */
LOOP_END;
          P5 = [P4];
LOOPLEZ LC0 = P5;
  /* loop body only executed if count
     is initially positive */
LOOP_END;  P5 = [P4];
LOOP LC1 = P5;
  /* loop body executed at least once */
LOOP_END;
P5 = [P4];
LOOPLEZ LC0 = P5;
  /* loop body only executed if count
     is initially positive */
LOOP_END;
```

The assembler translates LOOP, LOOPZ and LOOPLEZ instructions to LSETUP, LSETUPZ and LSETUPLEZinstructions which contain the PC-relative address of the final instruction in the loop. The

LOOP_ENDpseudo-instruction is simply there to locate the end of the loop so it does not get translated to any instruction at all. On disassembly the replacement LSETUP-type instruction is seen.

Two sets of zero-overhead loop registers implement loops, using hardware counters instead of software instructions to evaluate loop conditions. After evaluation, processing branches to a new target address. Both sets of registers include the Loop Counter (LC), Loop Top (LT), and Loop Bottom (LB) registers.

The loop registers table describes the 32-bit loop register sets.

Table 4-8: Loop Registers

Registers	Description	Function
LC0, LC1	Loop Counters	Maintains a count of the remaining iterations of the loop
LT0, LT1	Loop Tops	Holds the address of the first instruction within a loop
LB0, LB1	Loop Bottoms	Holds the address of the last instruction of the loop

When an instruction at address X is executed, and X matches the contents of LB0, then the next instruction executed will be from the address in LT0. In other words, when PC == LB0, then an implicit jump to LT0 is executed.

The LC0 and LC1 registers are unsigned 32-bit registers, each supporting 2^{32} -1 iterations through the loop. When, LCx = 0, the loop is disabled and a single pass of the code executes.

A loopback only occurs when the count is greater than or equal to 2. If the count is nonzero, then the count is decremented by 1. For example, consider the case of a loop with two iterations. At the beginning, the count is 2. On reaching the first loop end, the count is decremented to 1 and the program flow jumps back to the top of the loop (to execute a second time). On reaching the end of the loop again, the count is decremented to 0, but no loopback occurs (because the body of the loop has already been executed twice).

The LSETUP, LSETUPZ or LSETUPLEZinstructions can be used to load all three registers of a loop unit at once. When executing one of these loop setup instructions the program sequencer loads the address of the loop's last instruction into LBx and the address of the loop's first instruction into LTx. The bottom address of the loop is computed from a PC-relative offset held in the instruction, which limits the maximum loop size to 2046 bytes. It is recommended that the loop top address is the instruction after the loop setup instruction.

Each loop register can also be loaded individually with a register transfer, but this incurs a significant overhead if the loop count is nonzero (the loop is active) at the time of the transfer.

For compatibility with earlier Blackfin processors, the LSETUP instruction without immediate count may contain a start offset. With this form of the instruction the loop top can be to be up to 30 bytes after the loop setup. However a four-cycle latency occurs on the first loopback if the LSETUP specifies a nonzero start offset.

A legacy form of the LOOP syntax is supported. In this syntax a loop gets assigned a name. All loop instructions are enclosed between the LOOP_BEGIN and LOOP_END brackets.

```
Legacy named LOOP syntax
  LC0 = R0;
  LOOP myloop LC0;
/* instructions between setup and body
     NOT RECOMMENDED */
  LOOP_BEGIN myloop;
    /* loop body */;
  LOOP_END myloop;
```

The processor supports a four-location instruction loop buffer that reduces instruction fetches while in loops. If the loop code contains four or fewer instructions, then no fetches to instruction memory are necessary for any number of loop iterations, because the instructions are stored locally. The loop buffer effectively eliminates the instruction fetch time in loops with more than four instructions by allowing fetches to take place while instructions in the loop buffer are being executed.

The processor has no restrictions regarding which instructions can occur in a loop end position. Branches and calls are allowed in that position.

Two-Dimensional Loops

The processor features two loop units. Each provides its own set of loop registers.

- LC[1:0] – the Loop Count registers
- LT[1:0] – the Loop Top address registers
- LB[1:0] – the Loop Bottom address registers

Therefore, two-dimensional loops are supported directly in hardware, consisting of an outer loop and a nested inner loop.

NOTE: The outer loop is always represented by loop unit 0 (LC0, LT0, LB0) while loop unit 1 (LC1, LT1, LB1) manages the inner loop.

To enable the two nested loops to end at the same instruction (LB1 equals LB0), loop unit 1 is assigned higher priority than loop unit 0. A loopback caused by loop unit 1 on a particular instruction (PC==LB1, LC1>=2) will prevent loop unit 0 from looping back on that same instruction, even if the address matches. Loop unit 0 is allowed to loop back only after the loop count 1 is exhausted. Consequently when no instructions appear after the inner loop within the outer loop body, the outer loop must use LC0 and the inner loop LC1. The following example shows a two-dimensional loop.

```
#define M 32
#define N 1024
  P4 = M (Z);
  P5 = N-1 (Z);
  LOOP LC0 = P4;
    R7 = 0 ;
```

```
    MNOP || R2 = [I0++] || R3 = [I1++] ;
    LOOP LC1 = P5;
      R5 = R2 + R3 (NS) || R2 = [I0] || R3 = [I1++] ;
      R7 = R5 + R7 (NS) || [I0++] = R5;
    LOOP_END ;
    R5 = R2 + R3 ;
    R7 = R5 + R7 (NS) || [I0++] = R5 ;
    [I2++] = R7 ;
  LOOP_END ;
```

The example processes an M by N data structure. The inner loop is unrolled and passes only N-1 times. The outer loop is not unrolled and still provides room for optimization.

Loop Unrolling

Typical DSP algorithms are coded for speed rather than for small code size. Especially when fetching data from circular buffers, loops are often unrolled in order to pass only N-1 times. The initial data fetch is executed before the loop is entered. Similarly, the final calculations are done after the loop terminates, for example:

```
#define N 1024
global_setup:
  I0.H = 0xFF80; I0.L = 0x0000; B0 = I0; L0 = N*2 (Z);
  I1.H = 0xFF90; I1.L = 0x0000; B1 = I1; L1 = N*2 (Z);
  P5 = N-1 (Z);

algorithm:
  A0 = 0 || R0.H = W[I0++] || R1.L = W[I1++];
  LOOP LC0 = P5;
    A0+= R0.H * R1.L || R0.H = W[I0++] || R1.L = W[I1++];
  LOOP_END;
  A0+= R0.H * R1.L;
```

This technique has the advantage that data is fetched exactly N times and the I-Registers have their initial value after processing. The "algorithm" sequence can be executed multiple times without any need to initialize DAG-Registers again.

Saving and Resuming Loops

Normally, loops can process and terminate without regard to system-level concepts. Even if interrupted by interrupts or exceptions, no special care is needed. There are, however, a few situations that require special attention-whenever a loop is interrupted by events that require the loop resources themselves, that is:

- If the loop is interrupted by an interrupt service routine that also contains a hardware loop and requires the same loop unit
- If the loop is interrupted by a preemptive task switch

- If the loop contains a CALL instruction that invokes an unknown subroutine that may have local loops

In scenarios like these, the loop environment can be saved and restored by pushing and popping the loop registers. For example, to save Loop Unit 0 onto the system stack, use this code:

```
[--SP] = LC0;
[--SP] = LB0;
[--SP] = LT0;
```

To restore Loop Unit 0 from system stack, use:

```
LT0 = [SP++];
LB0 = [SP++];
LC0 = [SP++];
```

It is obvious that writes or pops to the loop registers cause some internal side effects to re-initialize the loop hardware properly. The hardware does not force the user to save and restore all three loop registers, as there might be cases where saving one or two of them is sufficient. Consequently, every pop instruction in the example above may require the loop hardware to re-initialize again. This takes multiple cycles, as the loop buffers must also be prefilled again.

To avoid unnecessary penalty cycles, the loop hardware follows these rules:

- Restoring LC0 and LC1 registers always re-initializes the loop hardware and causes a ten-cycle "replay" penalty.
- Restoring LT0, LT1, LB0, and LB1 performs in a single cycle if the respective loop counter register is zero.
- If LCx is non-zero, every write to the LTx and LBx registers also attempts to re-initialize the loop hardware and causes a ten-cycle penalty.

In terms of performance, there is a difference depending on the order that the loop registers are popped. For best performance, restore the LCx registers last. Furthermore, it is recommended that interrupt service routines and global subroutines that contain hardware loops terminate their local loops cleanly, that is, do not artificially break the loops and do not execute return instructions within their loops. This guarantees that the LCx registers are 0 when LTx and LBx registers are popped.

Example Code for Using Hardware Loops in an ISR

The following code shows the optimal method of saving and restoring when using hardware loops in an interrupt service routine.

```
Saving and Restoring With Hardware Loops
lhandler:
<Save other registers here>
[--SP] = LC0; /* save loop 0 */
[--SP] = LB0;
[--SP] = LT0;

/* ... Handler code here ... */
```

```
/* If the handler uses loop 0, it is a good idea to have
it leave LC0 equal to zero at the end. Normally, this will
happen naturally as a loop is fully executed. If LC0 == 0,
then LT0 and LB0 restores will not incur additional cycles.
If LC0 != 0 when the following pops happen, each pop will
incur a ten-cycle "replay" penalty. Popping or writing LC0
always incurs the penalty. */

LT0 = [SP++];
LB0 = [SP++];
LC0 = [SP++];   /* This will cause a "replay," that is, a ten-cycle refetch. */

/* ... Restore other registers here ... */

RTI;
```

Events and Interrupts

The Event Controller of the processor manages five types of activities or events:

- Emulation
- Reset
- Nonmaskable interrupts (NMI)
- Exceptions
- Interrupts

Note the word *event* describes all five types of activities. The Event Controller manages fifteen different events in all: Emulation, Reset, NMI, Exception, and eleven Interrupts.

An *interrupt* is an event that changes normal processor instruction flow and is asynchronous to program flow. In contrast, an *exception* is a software initiated event whose effects are synchronous to program flow.

The event system is nested and prioritized. Consequently, several service routines may be active at any time, and a low priority event may be preempted by one of higher priority.

The processor employs a two-level event control mechanism. The processor System Event Controller (SEC) works with the Core Event Controller (CEC) to prioritize and control all system interrupts. The SEC provides mapping between the many peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core. Individual interrupt sources can be masked in the SEC.

The CEC supports nine general-purpose interrupts (`IVG7 - IVG15`) in addition to the dedicated interrupt and exception events that are described in the table. It is recommended that at least the two lowest priority interrupts (`IVG14` and `IVG15`) be reserved for software interrupt handlers, leaving seven prioritized interrupt inputs (`IVG7 - IVG13`) to support the system. The SEC described in this manual maps all system events

to IVG11 leaving IVG12 to IVG15 free for use as software interrupt handlers. Refer to the Hardware Reference Manual for your processor for a detailed description of the SEC.

Table 4-9: Core Event Mapping

	Event Source	Core Event Name
Core Events	Emulation (highest priority)	EMU
	Reset	RST
	NMI	NMI
	Exception	EVX
	Reserved	-
	Hardware Error	IVHW
	Core Timer	IVTMR

Core Event Controller Registers

The Event Controller uses three core MMRs to coordinate pending event requests. In each of these MMRs, the 16 lower bits correspond to the 16 event levels (for example, bit 0 corresponds to "Emulator mode"). The registers are:

- IMASK - interrupt mask
- ILAT - interrupt latch
- IPEND - interrupts pending

These three registers are accessible in Supervisor mode only.

IMASK Register

The Core Interrupt Mask register (IMASK) indicates which interrupt levels are allowed to be taken. The IMASK register may be read and written in Supervisor mode. Bits [15:5] have significance; bits [4:0] are hard-coded to 1 and events of these levels are always enabled. If IMASK[N] == 1 and ILAT[N] == 1, then interrupt N will be taken if a higher priority is not already recognized. If IMASK[N] == 0, and ILAT[N] gets set by interrupt N, the interrupt will not be taken, and ILAT[N] will remain set. For more information, see the Interrupt Mask Register.

ILAT Register

Each bit in the Core Interrupt Latch register (ILAT) indicates that the corresponding event is latched, but not yet accepted into the processor. The bit is reset before the first instruction in the corresponding ISR is executed. At the point the interrupt is accepted, ILAT[N] will be cleared and IPEND[N] will be set simultaneously. The ILAT register can be read in Supervisor mode. Writes to ILAT are used to clear bits only (in

Supervisor mode). To clear bit N from ILAT, first make sure that IMASK[N] == 0, and then write ILAT[N] = 1. This write functionality to ILAT is provided for cases where latched interrupt requests need to be cleared (canceled) instead of serviced. For more information, see the Interrupt Latch Register.

The RAISE instruction can be used to set ILAT[15] through ILAT[5], and also ILAT[2] or ILAT[1].

Only the JTAG TRST pin can clear ILAT[0].

IPEND Register

The Core Interrupt Pending register(IPEND) keeps track of all currently nested interrupts. Each bit in IPEND indicates that the corresponding interrupt is currently active or nested at some level. It may be read in Supervisor mode, but not written. The IPEND[4] bit is used by the Event Controller to temporarily disable interrupts on entry and exit to an interrupt service routine. For more information, see the Interrupt Pending Register.

When an event is processed, the corresponding bit in IPEND is set. The least significant bit in IPEND that is currently set indicates the interrupt that is currently being serviced. At any given time, IPEND holds the current status of all nested events.

Event Vector Table

The Event Vector Table (EVT) is a hardware table with sixteen entries that are each 32 bits wide. The EVT contains an entry for each possible core event. Entries are accessed as MMRs, and each entry can be programmed at reset with the corresponding vector address for the interrupt service routine. When an event occurs, instruction fetch starts at the address location in the EVT entry for that event.

The processor architecture allows unique addresses to be programmed into each of the interrupt vectors; that is, interrupt vectors are not determined by a fixed offset from an interrupt vector table base address. This approach minimizes latency by not requiring a long jump from the vector table to the actual ISR code.

The table lists events by priority. Each event has a corresponding bit in the event state registers ILAT, IMASK, and IPEND.

Table 4-10: Core Event Vector Table

Name	Event Class	Event Vector Register	MMR Location	Notes
EMU	Emulation	EVT0	0x1FE0 2000	Highest priority. Vector address is provided by JTAG.
RST	Reset	EVT1	0x1FE0 2004	RAISE 1 vector.Not used by Reset Control Unit (RCU).
NMI	NMI	EVT2	0x1FE0 2008	
EVX	Exception	EVT3	0x1FE0 200C	
Reserved	Reserved	EVT4	0x1FE0 2010	Reserved vector

Table 4-10: Core Event Vector Table (Continued)

Name	Event Class	Event Vector Register	MMR Location	Notes
IVHW	Hardware Error	EVT5	0x1FE0 2014	
IVTMR	Core Timer	EVT6	0x1FE0 2018	
IVG7	Interrupt 7	EVT7	0x1FE0 201C	Reserved System interrupt
IVG8	Interrupt 8	EVT8	0x1FE0 2020	Reserved System interrupt
IVG9	Interrupt 9	EVT9	0x1FE0 2024	Reserved System interrupt
IVG10	Interrupt 10	EVT10	0x1FE0 2028	Reserved System interrupt
IVG11	Interrupt 11	EVT11	0x1FE0 202C	System interrupt for System Event Controller (SEC)
IVG12	Interrupt 12	EVT12	0x1FE0 2030	Software or system interrupt
IVG13	Interrupt 13	EVT13	0x1FE0 2034	Software or system interrupt
IVG14	Interrupt 14	EVT14	0x1FE0 2038	Software or system interrupt
IVG15	Interrupt 15	EVT15	0x1FE0 203C	Software interrupt

Return Registers and Instructions

Similarly to the RETS register controlled by CALL and RTS instructions, interrupts and exceptions also use single-entry hardware stack registers. If an interrupt is serviced, the program sequencer saves the return address into the RETI register prior to jumping to the event vector. A typical interrupt service routine terminates with an RTI instruction that instructs the sequencer to reload the Program Counter, PC, from the RETI register. The following example shows a simple interrupt service routine.

```
isr:
  [--SP] = (R7:0, P5:0);  /* push core registers */
  [--SP] = ASTAT;         /* push arithmetic status */

  /* place core of service routine here */

  ASTAT = [SP++];          /* pop arithmetic status */
  (R7:0, P5:0) = [SP++];   /* pop core registers */
  RTI;                     /* return from interrupt */
isr_end:
```

There is no need to manage the RETI register when interrupt nesting is not enabled. If however, nesting is enabled and the respective service routine must be interruptible by an interrupt of higher priority, the RETI register must be saved, most likely onto the stack.

Instructions that access the RETI register do have an implicit side effect-reading the RETI register enables interrupt nesting. Writing to it disables nesting again. This enables the service routine to break itself down into interruptible and non-interruptible sections. For example:

```
isr:
  [--SP] = (R7:0, P5:0);  /* push core registers */
  [--SP] = ASTAT;         /* push arithmetic status */

  /* place critical or atomic code here */

  [--SP] = RETI;          /* enable nesting */

  /* place core of service routine here */

  RETI = [SP++];          /* disable nesting */

  /* more critical or atomic instructions */

  ASTAT = [SP++];         /* pop arithmetic status */
  (R7:0, P5:0) = [SP++]   /* pop core registers */
  RTI;                    /* return from interrupt */
isr.end:
```

If there is not a need for non-interruptible code inside the service routine, it is good programming practice to enable nesting immediately. This avoids unnecessary delay to high priority interrupt routines. For example:

```
isr:
  [--SP] = RETI;          /* enable nesting */
  [--SP] = (R7:0, P5:0);  /* push core registers */
  [--SP] = ASTAT;         /* push arithmetic status */

  /* place core of service routine here */

  ASTAT = [SP++];         /* pop arithmetic status */
  (R7:0, P5:0) = [SP++];  /* pop core registers */
  RETI = [SP++];          /* disable nesting */
  RTI;                    /* return from interrupt */
isr.end:
```

See Nesting of Interrupts for more details on interrupt nesting.

Emulation Events, NMI, and Exceptions use a technique similar to "normal" interrupts. However, they have their own return register and return instruction counterparts. The table provides an overview.

Table 4-11: Return Registers and Instructions

Name	Event Class	Return Register	Return Instruction
EMU	Emulation	RETE	RTE
RST	Reset	-	-

Table 4-11: Return Registers and Instructions (Continued)

Name	Event Class	Return Register	Return Instruction
NMI	NMI	RETN	RTN
EVX	Exception	RETX	RTX
Reserved	Reserved	-	-
IVHW	Hardware Error	RETI	RTI
IVTMR	Core Timer	RETI	RTI
IVG7	Interrupt 7	RETI	RTI
IVG8	Interrupt 8	RETI	RTI
IVG9	Interrupt 9	RETI	RTI
IVG10	Interrupt 10	RETI	RTI
IVG11	Interrupt 11	RETI	RTI
IVG12	Interrupt 12	RETI	RTI
IVG13	Interrupt 13	RETI	RTI
IVG14	Interrupt 14	RETI	RTI
IVG15	Interrupt 15	RETI	RTI

Executing RTX, RTN, or RTE in a Lower Priority Event

Instructions `RTX`, `RTN`, and `RTE` are designed to return from an exception, NMI, or emulator event, respectively. Do not use them to return from a lower priority event. To return from an interrupt, use the `RTI` instruction. Failure to use the correct instruction may produce unintended results.

In the case of `RTX`, bit `IPEND[3]` is cleared. In the case of `RTI`, the bit of the highest priority interrupt in `IPEND` is cleared.

Emulation Interrupt

An emulation event causes the processor to enter Emulation mode, where instructions are read from the JTAG interface. It is the highest priority interrupt to the core.

For detailed information about emulation, see the Blackfin Processor Debug chapter of the *Blackfin Processor Hardware Reference* for your part.

Reset Interrupt

The reset control unit (RCU) controls how the core enters and exits reset and supplies the software address to which the core vectors once it leaves the reset state. See the Hardware Reference Manual for a detailed description of the RCU.

Executing RAISE 1 does not assert core reset directly. It simply creates an interrupt at level 1.

Use of EVT1 for the RAISE 1 interrupt vector is enabled by clearing bit 15 of the EVT_OVERRIDE register. When this bit is set the address supplied by the RCU is used as the vector location.

A reset signaled by the RCU always vectors to the address supplied by the RCU.

RAISE 1 does not save the return address in a register, so it is not possible to return from a level 1 interrupt.

In earlier Blackfin processors RAISE 1 caused a software reset. It is generally unsafe for a core to transfer control to boot code which assumes the core and system is coming out of reset when in fact nothing has been reset.

If RAISE 1 software reset functionality is desired in the application, the following software control is assumed for standard operation:

- After booting, EVT1 is programmed with the ISR location for software interrupt level 1.
- Bit 15 of EVT_OVERRIDE is cleared.
- When a RAISE 1 is executed:
  - The ISR goes through the appropriate mechanisms via the RCU to shut off all core interfaces.
  - The RCU then resets the core. This is seen by the core as an external reset.

NMI (Nonmaskable Interrupt)

The NMI entry is reserved for nonmaskable interrupts, which may be triggered by the system or by the core in response to parity errors.

Propagation of NMI from sources external to the core is under control of the system event controller (SEC). See the Hardware Reference manual for details of the SEC. The SYSNMI bit in the SEQSTATwill be set on vectoring to the NMI handler if triggered from a source external to the core.

See MEMORY CHAPTER for details of Parity errors. One of the parity error indicator bits in SEQSTAT (PEIC, PEDC, PEIX or PEDX) will be set on entry to the NMI handler if triggered by a parity error.

An level 2 interrupt may also be triggered by the RAISE 2 instruction.

An external NMI while the processor is an event handler that is already servicing an NMI, reset or emulation event will trigger an Unhandled NMI error system interrupt handled by the SEC.

If an exception occurs in an event handler that is already servicing an exception, NMI, or reset event, this will trigger a double fault system interrupt handled by the SEC. The core stalls until system intervention.

The NSPECABT bit in SEQSTAT will be set on entry to an NMI, reset or emulation handler if a non-speculative access such as a system MMR read, or a read from I/O Device memory was aborted by the event. The read will be reattempted on return from the handler which may not be desired if the read has side-effects, such as access to a FIFO.

Exceptions

Exceptions are discussed in Hardware Errors and Exception Handling.

Hardware Error Interrupt

Hardware Errors are discussed in Hardware Errors and Exception Handling.

Core Timer Interrupt

The Core Timer Interrupt (IVTMR) is triggered when the core timer value reaches zero. For more information about the core timer, see the hardware reference for your processor.

General-purpose Core Interrupts (IVG7-IVG15)

The System Event Controller (SEC) can forward interrupt requests to the core events IVG7-IVG15, referred to as general-purpose core interrupts. See Servicing System Interrupts.

Software can also trigger general-purpose interrupts by using the RAISE instruction. The RAISE instruction forces events for interrupts IVG15-IVG7, IVTMR, IVHW, NMI, and RST, but not for exceptions and emulation (EVX and EMU, respectively).

NOTE: It is a useful practice to reserve the two lowest priority interrupts (IVG15 and IVG14) for software interrupt handlers.

For system interrupts available on specific Blackfin processors, see the hardware reference for that processor.

Interrupt Processing

The following sections describe interrupt processing.

Global Enabling/Disabling of Interrupts

General-purpose interrupts can be globally disabled with the CLI Dreg instruction and re-enabled with the STI Dreg instruction, both of which are only available in Supervisor mode. Reset, NMI, emulation, and exception events cannot be globally disabled. Globally disabling interrupts clears IMASK[15:5] after saving IMASK's current state.

```
CLI R5;    /* save IMASK to R5 and mask all */
/* place critical instructions here */
STI R5;    /* restore IMASK from R5 again */
```

For more information about enabling and disabling interrupts, see the external event management chapter.

When multiple instructions need to be atomic or are too time-critical to be delayed by an interrupt, disable the general-purpose interrupts, but be sure to re-enable them at the conclusion of the code sequence.

Servicing Interrupts

The Core Event Controller (CEC) has a single interrupt queuing element per event—a bit in the `ILAT` register. The appropriate `ILAT` bit is set when an interrupt rising edge is detected (which takes two core clock cycles) and cleared when the respective `IPEND` register bit is set. The `IPEND` bit indicates that the event vector has entered the core pipeline. At this point, the CEC recognizes and queues the next rising edge event on the corresponding interrupt input. The minimum latency from the rising edge transition of the general-purpose interrupt to the `IPEND` output assertion is three core clock cycles. However, the latency can be much higher, depending on the core's activity level and state.

To determine when to service an interrupt, the controller logically AND's the three quantities in `ILAT`, `IMASK`, and the current processor priority level.

Servicing the highest priority interrupt involves these actions:

1. The interrupt vector in the Event Vector Table (EVT) becomes the next fetch address.

   On an interrupt, most instructions currently in the pipeline are aborted. On a service exception, all instructions after the excepting instruction are aborted. On an error exception, the excepting instruction and all instructions after it are aborted.

2. The return address is saved in the appropriate return register.

   The return register is `RETI` for interrupts, `RETX` for exceptions, `RETN` for NMIs, and `RETE` for debug emulation. The return address is the address of the instruction after the last instruction executed from normal program flow.

3. Processor mode is set to the level of the event taken.

   If the event is an NMI, exception, or interrupt, the processor mode is Supervisor. If the event is an emulation exception, the processor mode is Emulation.

4. Before the first instruction starts execution, the corresponding interrupt bit in `ILAT` is cleared and the corresponding bit in `IPEND` is set.

   Bit `IPEND[4]` is also set to disable all interrupts until the return address in `RETI` is saved.

Nesting of Interrupts

Interrupts are handled either with or without nesting, individually. For more information, see Return Registers and Instructions.

Non-nested Interrupts

If interrupts do not require nesting, all interrupts are disabled during the interrupt service routine. Note, however, that emulation, NMI, and exceptions are still accepted by the system.

When the system does not need to support nested interrupts, there is no need to store the return address held in `RETI`. Only the portion of the machine state used in the interrupt service routine must be saved in the Supervisor stack. To return from a non-nested interrupt service routine, only the `RTI` instruction must be executed, because the return address is already held in the `RETI` register.

The figure shows an example of interrupt handling where interrupts are globally disabled for the entire interrupt service routine.

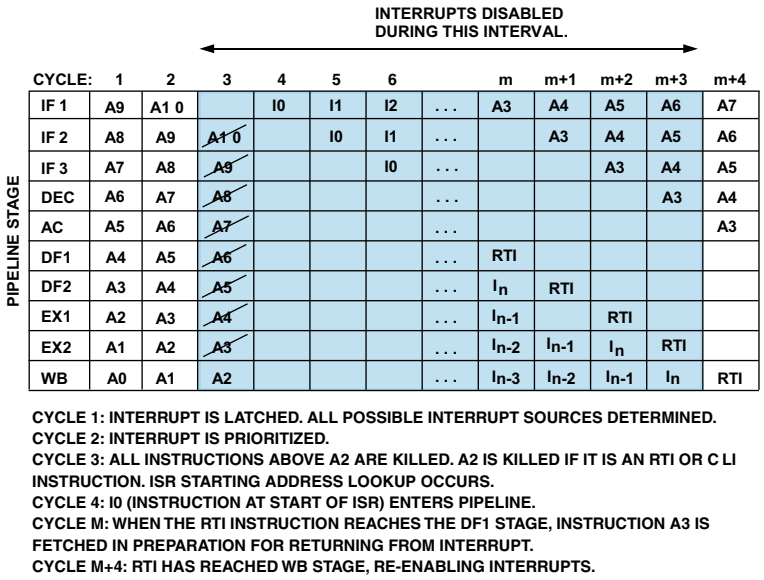


Figure 4-4: Non-nested Interrupt Handling

Nested Interrupts

If interrupts require nesting, the return address to the interrupted point in the original interrupt service routine must be explicitly saved and subsequently restored when execution of the nested interrupt service routine has completed. The first instruction in an interrupt service routine that supports nesting must save the return address currently held in `RETI` by pushing it onto the Supervisor stack (`[--SP] = RETI`). This clears the global interrupt disable bit `IPEND[4]`, enabling interrupts. Next, all registers that are modified

by the interrupt service routine are saved onto the Supervisor stack. Processor state is stored in the Supervisor stack, not in the User stack. Hence, the instructions to push RETI ([--SP] = RETI) and pop RETI (RETI = [SP++]) use the Supervisor stack.

The figure illustrates that by pushing RETI onto the stack, interrupts can be re-enabled during an interrupt service routine, resulting in a short duration where interrupts are globally disabled.

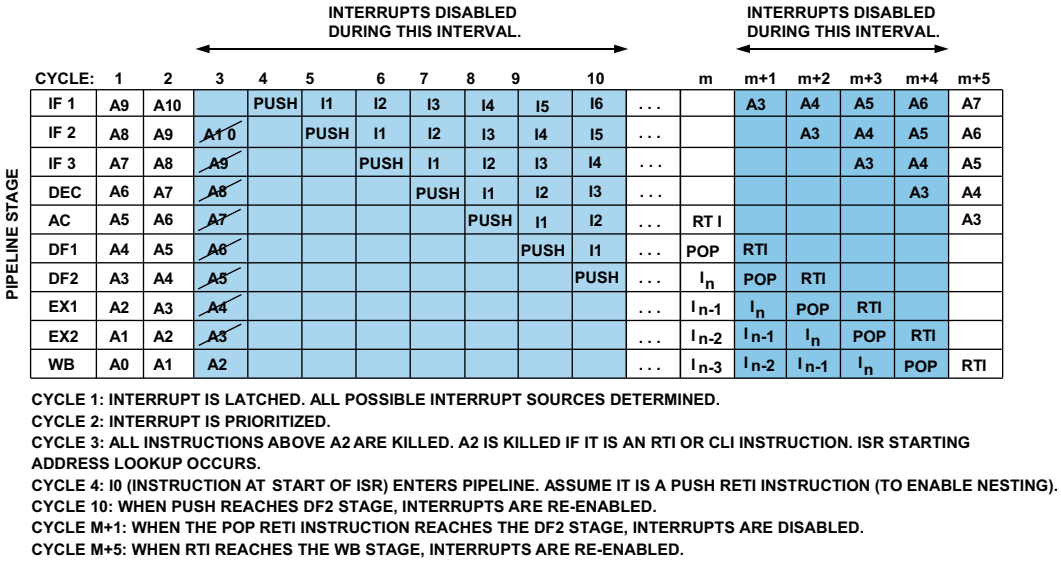


Figure 4-5: Nested Interrupt Handling

Example Prolog Code for Nested Interrupt Service Routine

```
Prolog Code for Nested ISR
/* Prolog code for nested interrupt service routine.
Push return address in RETI into Supervisor stack, ensuring that
interrupts are back on. Until now, interrupts have been
suspended.*/

ISR:
  [--SP] = RETI ; /* Enables interrupts and saves return
                  address to stack */
  [--SP] = ASTAT ;
  [--SP] = FP ;
  [-- SP] = (R7:0, P5:0) ;
/* Body of service routine. Note
none of the processor resources (accumulators, DAGs, loop counters
and bounds) have been saved. It is assumed this interrupt service
routine does not use the processor resources. */
```

Example Epilogue Code for Nested Interrupt Service Routine

```
Epilogue Code for Nested ISR
/* Epilog code for nested interrupt service routine.
Restore ASTAT, Data and Pointer registers. Popping RETI from Supervisor stack ensures
that interrupts are suspended between load of return address and RTI. */
(R7:0, P5:0) = [SP++] ;
FP    = [SP++] ;
ASTAT = [SP++] ;
RETI  = [SP++] ;
/* Execute RTI, which jumps to return address, re-enables interrupts, and switches to
User mode if this is the last nested interrupt in service. */
RTI;
```

The `RTI` instruction causes the return from an interrupt. The return address is popped into the `RETI` register from the stack, an action that suspends interrupts from the time that `RETI` is restored until `RTI` finishes executing. The suspension of interrupts prevents a subsequent interrupt from corrupting the `RETI` register.

Next, the `RTI` instruction clears the highest priority bit that is currently set in `IPEND`. The processor then jumps to the address pointed to by the value in the `RETI` register and re-enables interrupts by clearing `IPEND[4]`.

Self-Nesting of Core Interrupts

Interrupts that are "self-nested" can be interrupted by events at the same priority level. When the `SNEN` bit of the `SYSCFG` register is set, self-nesting of core interrupts is supported. Self-nesting is supported for any interrupt level generated with the `RAISE` instruction, as well as for core level interrupts.

As an example, assume that the `SNEN` bit is set and the processor is servicing an interrupt generated by the `RAISE 14;` instruction. Once the `RETI` register has been saved to the stack within the service routine, a second `RAISE 14;` instruction would allow the processor to service the second interrupt.

Self-nesting is not supported for system level peripheral interrupts such as the SPORT or SPI.

The `SYSCFG` register is discussed in the processor hardware reference.

Servicing System Interrupts

System event management is the responsibility of the System Event Controller (SEC). The SEC manages the configuration of each system interrupt or fault source and provides notification and identification of the highest priority active system interrupt request to each core. The SEC must be configured to notify the core of any particular system event. See the Processor Hardware Reference for a detailed description of the SEC.

The SEC prioritizes system interrupts and provides a single interrupt indicator to the core along with an interrupt ID. System interrupts are directed to core interrupt level 11. The interrupt ID is latched in the `CEC_SID` register for use in the interrupt service routine. For more information, see Context ID Register.

All the interrupts from the many peripherals and other components of the system are therefore handled by the single ISR for IVG11 which interacts with the SEC as well as performing the action appropriate to the original source of the interrupt.

The SEC prioritizes system interrupts. It will latch a new IVG11 interrupt if a higher priority system interrupt occurs after the current system interrupt has been acknowledged. Self-nesting of core interrupts must be enabled if the higher priority interrupt detected by the SEC is to be serviced by the core while it is servicing a lower priority system interrupt. In other words the SNEN bit of the SYSCFG register must be set.

Device specific code may be separated from general SEC programming concerns by maintaining a table of routines dedicated to each interrupt source. The IVG11 ISR is responsible for the following actions.

1. Reading the SEQ_SID MMR to obtain the interrupt source ID (SID).
2. Writing SEQ_SID to acknowledge the SEC.
3. Calling a device specific handler.
4. Writing SID to the SEC Global End Register MMR to indicate the interrupt has been serviced.

The following code is an example of an ISR which performs these actions which illustrates some programing concerns to be aware of.

```
ivg11_sec_isr:
   [--SP] = (P4:P5) ;
   P5 = [CEC_SID] ;
   /* Writing any value to CEC_SID sends ACK to SEC.
      After acknowledgement the value in
      CEC_SID will change asynchronously if a higher-priority
      interrupt becomes active. So the only reliable record
      of current interrupt source is in P5 */
   [CEC_SID] = R0 ;
   /* The interrupt source is safely in P5 and the SEC
      will only interrupt with higher priority notification.
      So it is safe to re-enable core interrupts by clearing
      IPEND[4], which is a side effect of saving RETI. */
   [--SP] = RETI ;
   [--SP] = RETS ;
   /* Use interrupt source to index a table of
      handlers for each specific interrupt. */
   P4 = specific_handlers;
   P4 = P4 + (P5 << 2);
   P4 = [P4];
   CALL (P4);
   /* Assume the handler has preserved the interrupt source in P5.
   */
   RETS = [SP++] ;
   RETI = [SP++] ;
   /* Write interrupt source to SEC Global End register
      to indicate the core has finished servicing the
      interrupt.  The SEC may now raise lower priority
      interrupts so this should happen with core interrupts
```

```
   disabled via IPEND[4] to prevent the lower priority
   interrupt being services before the higher priority ISR
   has returned.  So do this after RETI has been restored
   which implicitly sets IPEND[4]. */
[REG_SEC0_END] = P5 ;
(P4:5) = [SP++] ;
RTI ;
```

In practice use of any device driver or interrupt support routine supplied by a third party will require the IVG11 ISR code it is designed to work with. Software designed to work with CrossCore® Embedded Studio requires the ISR included in the runtime library and automatically linked into your program.

Clearing Interrupt Requests

When the processor services a core event, it automatically clears the requesting bit in the `ILAT` register and no further action is required by the interrupt service routine.

Interrupts from peripherals are forwarded to the core by the SEC. The first level of the interrupt service routine handles the interaction with the SEC as described in Servicing System Interrupts.

It is important to understand that the SEC does not provide any interrupt acknowledgment feedback to the peripherals. The signaling peripheral does not release its level-sensitive request until it is explicitly instructed by software to do so. If the request is not cleared by software, the peripheral keeps requesting the interrupt, and on receiving the end of interrupt acknowledgment the SEC will immediately re-interrupt the core. This causes the core to vector to the service routine again as soon as the `RTI` instruction is executed.

Every software routine that services peripheral interrupts must clear the signaling interrupt request in the respective peripheral. The individual peripherals provide customized mechanisms for how to clear interrupt requests. See the hardware reference manual for the details for peripherals in your processor.

Software Interrupts

Software cannot set bits of the `ILAT` register directly, as writes to `ILAT` cause a write-1-to-clear (W1C) operation. Instead, use the `RAISE` instruction to set individual `ILAT` bits by software. It safely sets any of the `ILAT` bits without affecting the rest of the register.

```
RAISE 14;         /* fire software interrupt request */
```

The `RAISE` instruction must not be used to fire emulation events or exceptions, which are managed by the related `EMUEXCPT` and `EXCPT` instructions. For details, see the external event management chapter.

Often, the `RAISE` instruction is executed in interrupt service routines to degrade the interrupt priority. This enables less urgent parts of the service routine to be interrupted even by low priority interrupts.

```
isr7:        /* service routine for IVG7 */
...
/* execute high priority instructions here */
```

```
/* handshake with signalling peripheral */
RAISE 14;
RTI;
isr7.end:
isr14:     /* service routine for IVG14 */
...
/* further process event initiated by IVG7 */
RTI;
isr14.end:
```

The example above may read data from any receiving interface, post it to a queue, and let the lower priority service routine process the queue after the `isr7` routine returns. Since `IVG15` is used for normal program execution in non-multi-tasking system, `IVG14` is often dedicated to software interrupt purposes.

The code in Example Code for an Exception Handler uses the same principle to handle an exception with normal interrupt priority level.

Latency in Servicing Events

In some processor architectures, if instructions are executed from external memory and an interrupt occurs while the instruction fetch operation is underway, then the interrupt is held off from being serviced until the current fetch operation has completed. Consider a processor operating at 300 MHz and executing code from external memory with 100 ns access times. Depending on when the interrupt occurs in the instruction fetch operation, the interrupt service routine may be held off for around 30 instruction clock cycles. When cache line fill operations are taken into account, the interrupt service routine could be held off for many hundreds of cycles.

In order for high priority interrupts to be serviced with the least latency possible, the processor allows any high latency fill operation to be completed at the system level, while an interrupt service routine executes from L1 memory. See the figure.

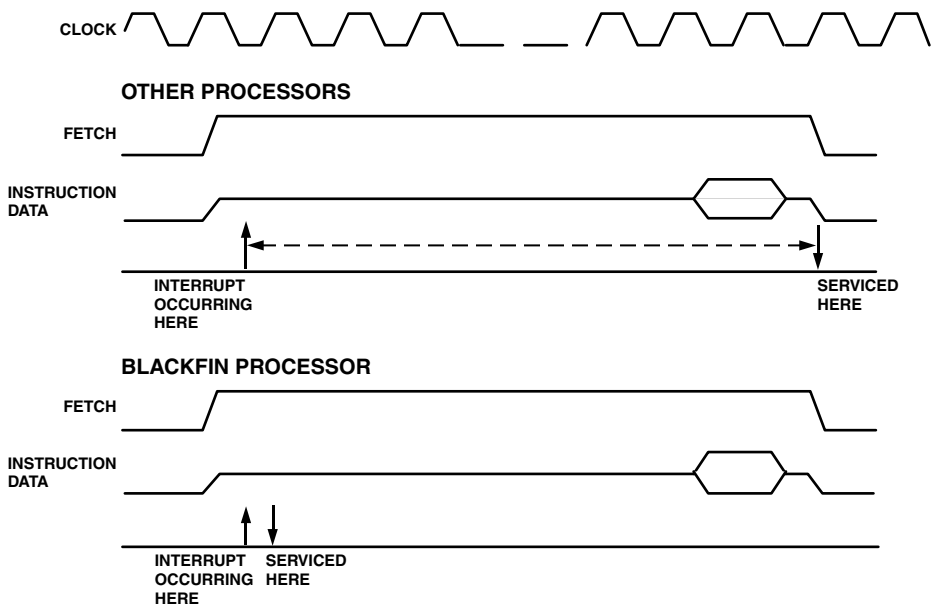


Figure 4-6: Minimizing Latency in Servicing an ISR

If an instruction load operation misses the L1 instruction cache and generates a high latency line fill operation, then when an interrupt occurs, it is not held off until the fill has completed. Instead, the processor executes the interrupt service routine in its new context, and the cache fill operation completes in the background.

Note the interrupt service routine must reside in L1 cache or SRAM memory and must not generate a cache miss, an L2 memory access, or a peripheral access, as the processor is already busy completing the original cache line fill operation. If a load or store operation is executed in the interrupt service routine requiring one of these accesses, then the interrupt service routine is held off while the original external access is completed, before initiating the new load or store.

If the interrupt service routine finishes execution before the load operation has completed, then the processor continues to stall, waiting for the fill to complete.

This same behavior is also exhibited for stalls involving reads of slow data memory or peripherals.

Writes to slow memory generally do not show this behavior, as the writes are deemed to be single cycle, being immediately transferred to the write buffer for subsequent execution.

For detailed information about cache and memory structures, see the memory chapter.

Hardware Errors and Exception Handling

The following sections describe hardware errors and exception handling.

SEQSTAT Register

The Sequencer Status register (SEQSTAT) contains information about the current state of the sequencer as well as diagnostic information from the last event. SEQSTAT is accessible only in Supervisor mode. For more information, see the Sequencer Status Register.

Hardware Error Interrupt

A hardware error interrupt indicates a hardware error or system malfunction. A core-related external hardware errors occurs when logic external to the core, such as a memory bus controller, is unable to complete a data transfer (read or write) initiated by the core and invokes a core hardware error interrupt on the SEC. This interrupt may be serviced by the core as described in the Servicing System Interrupts section.

An internal hardware error is generated by internal error conditions within the core, such as Performance Monitor overflow Such hardware errors invoke the internal hardware error interrupt (interrupt IVHW in the event vector table (EVT) and ILAT, IMASK, and IPEND registers). The hardware error interrupt service routine can then read the cause of the error from the 5-bit HWERRCAUSE field appearing in the sequencer status register (SEQSTAT) and respond accordingly.

The list of supported hardware conditions, with their related HWERRCAUSE codes, appears in the hardware conditions table. The bit code for the most recent error appears in the HWERRCAUSE field. If multiple hardware errors occur simultaneously, only the last one can be recognized and serviced. The core does not support prioritizing, pipelining, or queuing multiple error codes. The Hardware Error Interrupt remains active as long as any of the error conditions remain active.

Note that a hardware error status cannot be cleared by software.

In case of hardware error, the RETI does not store the address of the instruction that caused the hardware error. The error could have been caused by an instruction executed a number of core clock cycles before a hardware error is registered. In such scenarios, it is recommended to use the trace buffer to trace the instruction causing the hardware error.

Table 4-12: Hardware Conditions Causing Hardware Error Interrupts

Hardware Condition	HWERRCAUSE (Binary)	HWERRCAUSE (Hexadecimal)	Notes / Examples
Performance Monitor Overflow	0b10010	0x12	Refer to the performance monitor unit section of the processor hardware reference.
RAISE 5 instruction	0b11000	0x18	Software issued a RAISE 5 instruction to invoke the Hardware Error Interrupt (IVHW).
Reserved	All other bit combinations.	All other values.	

Exceptions (Events)

Exceptions are synchronous to the instruction stream. In other words, a particular instruction causes an exception when it attempts to finish execution. No instructions after the offending instruction are executed before the exception handler takes effect.

Many of the exceptions are memory related. For example, an exception is given when a cacheability protection lookaside buffer (CPLB) miss or protection violation occurs. Exceptions are also given when illegal instructions or illegal combinations of registers are executed.

An excepting instruction may or may not commit before the exception event is taken, depending on if it is a service type or an error type exception.

An instruction causing a service type event will commit, and the address written to the RETX register will be the next instruction after the excepting one. An example of a service type exception is the single step.

An instruction causing an error type event cannot commit, so the address written to the RETX register will be the address of the offending instruction. An example of an error type event is a CPLB miss.

NOTE: Usually the RETX register contains the correct address to return to. To skip over an excepting instruction, take care in case the next address is not simply the next linear address. This could happen when the excepting instruction is a loop end. In that case, the proper next address would be the loop top.

The EXCAUSE[5:0] field in the Sequencer Status register (SEQSTAT) is written whenever an exception is taken, and indicates to the exception handler which type of exception occurred. Refer to the events table for a list of events that cause exceptions.

ATTENTION: If an exception occurs in an event handler that is already servicing an exception, NMI, reset, or emulation event, this will trigger a double fault condition, and the address of the excepting instruction will be written to RETX.

Table 4-13: Events That Cause Exceptions

Exception	EXCAUSE [5:0]	Type: (E) Error (S) Service See Note 1.	Notes/Examples
Force Exception instruction EXCPT with 4-bit m field	m field	S	Instruction provides 4 bits of EXCAUSE.
Single step	0x10	S	When the processor is in single step mode, every instruction generates an exception. Primarily used for debugging.
Exception caused by a trace buffer full condition	0x11	S	The processor takes this exception when the trace buffer overflows (only when enabled by the Trace Unit Control register).

Table 4-13: Events That Cause Exceptions (Continued)

Exception	EXCAUSE [5:0]	Type: (E) Error (S) Service See Note 1.	Notes/Examples
Undefined instruction	0x21	E	May be used to emulate instructions that are not defined for a particular processor implementation.
Illegal instruction combination	0x22	E	See section for multi-issue rules in the *Blackfin Processor Programming Reference*.
Data access CPLB protection violation	0x23	E	Attempted read or write to Supervisor resource, or illegal data memory access. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions. (A simultaneous, dual access to two MMRs using the data address generators generates this type of exception.) In addition, this entry is used to signal a protection violation caused by disallowed memory access, and it is defined by the Memory Management Unit (MMU) cacheability protection lookaside buffer (CPLB).
Data access misaligned address violation	0x24	E	Attempted misaligned I/O or test data access.
Unrecoverable event	0x25	E	For example, an exception generated while processing a previous exception.
Data access CPLB miss	0x26	E	Used by the MMU to signal a CPLB miss on a data access.
Data access multiple CPLB hits	0x27	E	More than one CPLB entry matches data fetch address.
Exception caused by an emulation watchpoint match	0x28	E	There is a watchpoint match, and one of the EMUSW bits in the Watchpoint Instruction Address Control register (WPIACTL) is set.
Instruction fetch misaligned address violation	0x2A	E	Attempted misaligned instruction cache fetch. (Note this exception can never be generated from PC-relative branches, only from indirect branches.)
Instruction fetch CPLB protection violation	0x2B	E	Illegal instruction fetch access (memory protection violation).
Instruction fetch CPLB miss	0x2C	E	CPLB miss on an instruction fetch.
Instruction fetch multiple CPLB hits	0x2D	E	More than one CPLB entry matches instruction fetch address.

Table 4-13: Events That Cause Exceptions (Continued)

Exception	EXCAUSE [5:0]	Type: (E) Error (S) Service See Note 1.	Notes/Examples
Illegal use of supervisor resource	0x2E	E	Attempted to use a Supervisor register or instruction from User mode. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions.

NOTE: (1) For services (S), the return address is the address of the instruction that follows the exception. For errors (E), the return address is the address of the excepting instruction.

If an instruction causes multiple exception, the exception with the highest priority is first registered in the SEQSTAT. The exception priority is as listed in the exceptions by priority table. If the highest priority exception is handled, the next highest priority exception is registered and can be handled (and so on).

For example, suppose that the following instruction generates an instruction CPLB miss (0x2C) exception and a data CPLB miss (0x26) exception. On execution of this instruction, a instruction CPLB will be first generated. After this instruction exception is handled by the user the core will execute the instruction again and this time it will generate a data CPLB exception.

```
[P0] = R0 ;
/* generates an instruction CPLB miss and a data CPLB miss */
```

Table 4-14: Exceptions by Descending Priority

Priority	Exception	EXCAUSE
1	Unrecoverable Event	0x25
2	I-Fetch Multiple CPLB Hits	0x2D
3	I-Fetch Misaligned Access	0x2A
4	I-Fetch Protection Violation	0x2B
5	I-Fetch CPLB Miss	0x2C
6	I-Fetch Access Exception	0x29
7	Watchpoint Match	0x28
8	Undefined Instruction	0x21
9	Illegal Combination	0x22
10	Illegal Use of Protected Resource	0x2E
11	DAG0 Multiple CPLB Hits	0x27
12	DAG0 Misaligned Access	0x24

Table 4-14: Exceptions by Descending Priority (Continued)

Priority	Exception	EXCAUSE
13	DAG0 Protection Violation	0x23
14	DAG0 CPLB Miss	0x26
15	DAG1 Multiple CPLB Hits	0x27
16	DAG1 Misaligned Access	0x24
17	DAG1 Protection Violation	0x23
18	DAG1 CPLB Miss	0x26
19	EXCPT Instruction	m field
20	Single Step	0x10
21	Trace Buffer	0x11

Exceptions While Executing an Exception Handler

While executing the exception handler, avoid issuing an instruction that generates another exception. If an exception is caused while executing code within the exception handler, the NMI handler, or the reset vector:

- A double fault interrupt is sent to the SEC.
- The excepting instruction is not committed. All writebacks from the instruction are prevented.
- The `EXCAUSE` field in the `SEQSTAT` register is set to `0x25` (Unrecoverable Event) and `RETX` is updated with the address of the faulting instruction.
- The generated exception is not taken, and `PC` is not advanced.
- The core continues to fetch the same instruction and convert it to a nop until the exception goes away, or a higher priority interrupt takes over, or reset is asserted by the system.

In practice the only sensible means of recovery is to configure the SEC to reset the core on receipt of a double fault interrupt.

The `SEQSTAT` and `RETX` registers can be inspected within a debugger to diagnose the problem.

Allocating the System Stack

The software stack model for processing exceptions implies that the Supervisor stack must never generate an exception while the exception handler is saving its state. However, if the Supervisor stack is in cached or protected memory it may, in fact, cause CPLB miss exceptions.

One way to guarantee that the Supervisor stack never generates an exception is by calculating the maximum space that all interrupt service routines and the exception handler occupy while they are active, and then ensuring active CPLB entries cover this amount of memory. This may not be practical if the space required for the stack is large as only a limited number of CPLB entries may be active at once.

Another option is to provide a separate stack for the exception handler, as it is easier to calculate the total space required for this stack and to ensure it is covered by a single CPLB entry. The following code illustrates switching to a dedicated exception stack.

```
Switching stack within an exception handler
  .SECTION L1_scratchpad;
  .ALIGN 4;
  .VAR except_save_sp, except_stack[EXCEPT_STACK_SIZE];

  .SECTION L1_code;
except_handler:
  [ except_save_sp ] = SP;    /* save stack pointer */
  SP = except_stack+EXCEPT_STACK_SIZE;
  /* now safe to save registers in except_stack */
  [--SP] = (R7:6, P5:4);
  [--SP] = ASTAT;
  /* place core of service routine here */
  ASTAT = [SP++];
  (R7:6, P5:4) = [SP++];
  /* restore stack pointer before return */
  SP = [ except_save_sp ];
  RTX;
except_handler.end:
```

Similar considerations apply to parity error handlers. If the system stack could be in memory that caused the parity error then it is necessary to switch to a stack in a different memory region, such as ECC protected L2 memory, before saving any registers or risk a double parity error fault.

Exceptions and the Pipeline

Interrupts and exceptions treat instructions in the pipeline differently.

- When an interrupt occurs, all instructions in the pipeline are aborted.
- When an exception occurs, all instructions in the pipeline after the excepting instruction are aborted. For error exceptions, the excepting instruction is also aborted.

Because exceptions, NMIs, and emulation events have a dedicated return register, guarding the return address is optional. Consequently, the `PUSH` and `POP` instructions for exceptions, NMIs, and emulation events do not affect the interrupt system.

Note, however, the return instructions for exceptions (`RTX`, `RTN`, and `RTE`) do clear the Least Significant Bit (LSB) currently set in `IPEND`.

Deferring Exception Processing

Exception handlers are usually long routines, because they must discriminate among several exception causes and take corrective action accordingly. The length of the routines may result in long periods during which the interrupt system is, in effect, suspended.

To avoid lengthy suspension of interrupts, write the exception handler to identify the exception cause, but defer the processing to a low priority interrupt. To set up the low priority interrupt handler, use the Force Interrupt / Reset instruction (RAISE).

NOTE: When deferring the processing of an exception to lower priority interrupt IVGx, the system must guarantee that IVGx is entered before returning to the application-level code that issued the exception. If a pending interrupt of higher priority than IVGx occurs, it is acceptable to enter the high priority interrupt before IVGx.

Example Code for an Exception Handler

The following code is for an exception routine handler with deferred processing.

```
Exception Routine Handler With Deferred Processing
/* Determine exception cause by examining EXCAUSE field in SEQSTAT (first save contents
of R7, P5, P4 and ASTAT in Supervisor SP on a private stack.) */
.SECTION L1_scratchpad ;
#define EXCEPT_STACK_SZ 4
.VAR EXCEPT_SAVED_SP, EXCEPT_STACK[EXCEPT_STAK_SZ] ;
.SECTION L1_code;
except_handler:
[EXCEPT_SAVED_SP] = SP ;
SP = EXCEPT_STACK+EXCEPT_STACK_SZ ;
[--SP] = (R7,P5:4) ;
[--SP] = ASTAT ;
R7 = SEQSTAT ;
/* Mask the contents of SEQSTAT, and leave only EXCAUSE in R0 */
R7 <<= 26 ;
R7 >>= 26 ;
/* Using jump table EVTABLE, jump to the event pointed to by R0 */
P5 = R7 ;
P4 = _EVTABLE ;
P5 = P4 + ( P5 << 1 ) ;
R7 = W [ P5 ] (Z) ;
P4 = R0 ;
JUMP (PC + P4) ;
/* The entry point for an event is as follows. Here, processing is deferred to low
priority interrupt IVG12. Also, parameter passing would typically be done here. */
_EVENT1:
RAISE 12 ;
JUMP.S _EXIT ;
/* Entry for event at IVG13 */
_EVENT2:
RAISE 13 ;
JUMP.S _EXIT ;
/* Comments for other events */
/* At the end of handler, restore R7, P5, P1 and ASTAT, and return. */
_EXIT:
ASTAT = [SP++] ;
(R7,P5:4) = [SP++] ;
SP = [EXCEPT_SAVED_SP] ;
```

```
RTX ;
_EVTABLE:
.byte2 addr_event1;
.byte2 addr_event2;
...
.byte2 addr_eventN;
/* The jump table EVTABLE holds 16-bit address offsets for each event. With offsets,
this code is position independent and the table is small.
+--------------+
| addr_event1  | _EVTABLE
+--------------+
| addr_event2  | _EVTABLE + 2
+--------------+
|     . . .    |
+--------------+
| addr_eventN  | _EVTABLE + 2N
+--------------+
*/
```

Example Code for an Exception Routine

The following code provides an example framework for an interrupt routine jumped to from an exception handler such as that described above.

```
Interrupt Routine for Handling Exception
[--SP] = RETI ;   /* Push return address on stack. */

/* Put body of routine here.*/

RETI = [SP++] ;   /* To return, pop return address and jump. */

RTI ;   /* Return from interrupt. */
```

ADSP-BF70x Sequencer Related (REGFILE) Register Descriptions

Register File (REGFILE) contains the following registers.

Table 4-15: ADSP-BF70x REGFILE Register List

Name	Description
Pn	Pointer Register
SP	Stack Pointer Register
FP	Frame Pointer Register
ASTAT	Arithmetic Status Register

Table 4-15: ADSP-BF70x REGFILE Register List (Continued)

Name	Description
RETS	Return from Subroutine Register
LCn	Loop Count Register
LTn	Loop Top Register
LBn	Loop Bottom Register
CYCLES	Cycle Count (32 LSBs) Register
CYCLES2	Cycle Count (32 MSBs) Register
USP	User Stack Pointer Register
SEQSTAT	Sequencer Status Register
SYSCFG	System Configuration Register
RETI	Return from Interrupt Register
RETX	Return from Exception Register
RETN	Return from NMI Register
RETE	Return from Emulator Register

Pointer Register

The Pn registers (pointer register files) are 32 bits wide. Although pointer registers are primarily used for address calculations, these registers may also be used for general integer arithmetic with a limited set of arithmetic operations. For instance, pointer register math may be used to maintain counters. Unlike the data registers (Rn, pointer register arithmetic does not affect the arithmetic status (ASTAT) register status bits.

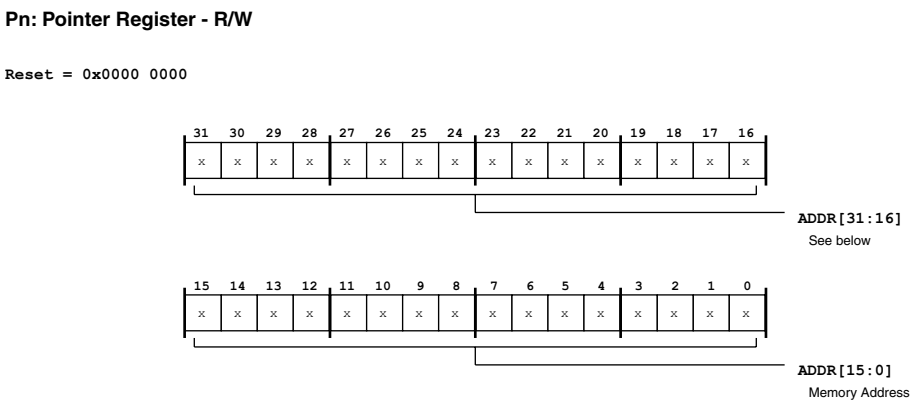


Figure 4-7: Pn Register Diagram

Table 4-16: Pn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Memory Address. The Pn.ADDR bit field (bits 0:31) holds either an address for address calculations or data for arithmetic operations.

Stack Pointer Register

In many respects, the stack and frame pointer registers perform like the other Pn registers. The SP and FP registers may act as general pointers in any of the load/store instructions. But, the SP and FP registers have additional functionality. To speed up context switching, there are two stack pointer registers, a user stack pointer (USP) and a supervisor stack pointer (SP). In assembly code syntax, one should always just use the SP syntax. Based on the current processor mode (supervisor or user), the correct register stack pointer will be used. For context switching, in supervisor mode, one may explicitly refer to the user stack pointer by using the USP syntax. Trying to use USP syntax in user mode causes an exception. Some load and store instructions use SP and FP implicitly. These instructions include LINK and UNLINK instructions, which control stack frame space and manage the SP register.

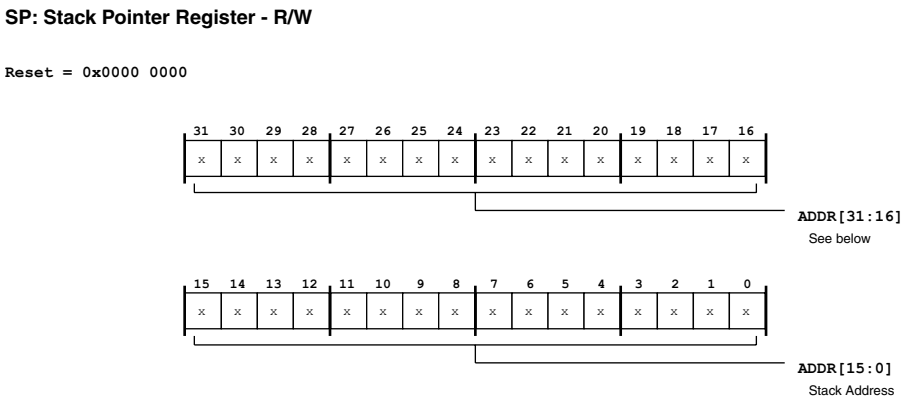


Figure 4-8: SP Register Diagram

Table 4-17: SP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Stack Address. The SP.ADDR bit field (bits 0:31) holds an address (pointer) for address calculations or stack operations.

Frame Pointer Register

In many respects, the stack and frame pointer registers perform like the other Pn registers. The SP and FP registers may act as general pointers in any of the load/store instructions. But, the SP and FP registers have additional functionality. To speed up context switching, there are two stack pointer registers, a user stack pointer (USP) and a supervisor stack pointer (SP). In assembly code syntax, one should always just use the SP syntax. Based on the current processor mode (supervisor or user), the correct register stack pointer will be used. For context switching, in supervisor mode, one may explicitly refer to the user stack pointer by using the USP syntax. Trying to use USP syntax in user mode causes an exception. Some load and store instructions use SP and FP implicitly. These instructions include LINK and UNLINK instructions, which control stack frame space and manage the SP register.

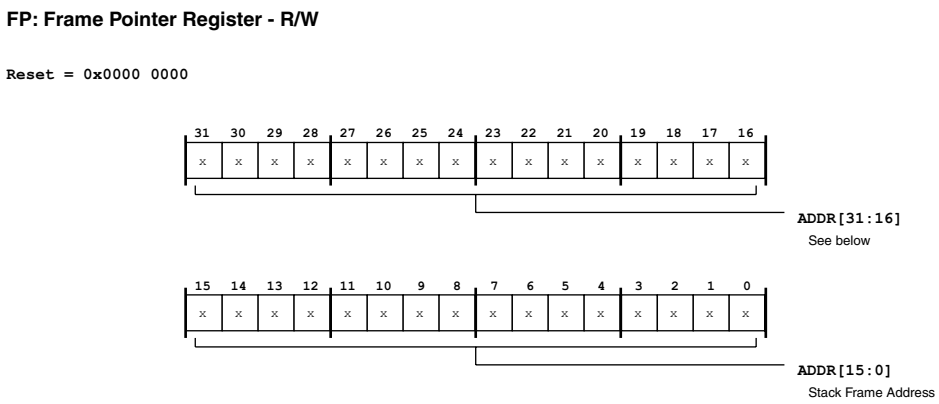


Figure 4-9: FP Register Diagram

Table 4-18: FP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Stack Frame Address. The FP.ADDR bit field (bits 0:31) hold either an address for address calculations or stack operations.

Arithmetic Status Register

The processor updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation. The ASTAT.AQ bit is updated, indicating the status of the most recent Divs or Divq instruction. The ASTAT.RND_MOD bit does not indicate status, rather this bit selects unbiased or biased rounding for operations supporting rounding.

If execution of an instruction generates status, the instruction's reference page indicates the affected arithmetic status bits.

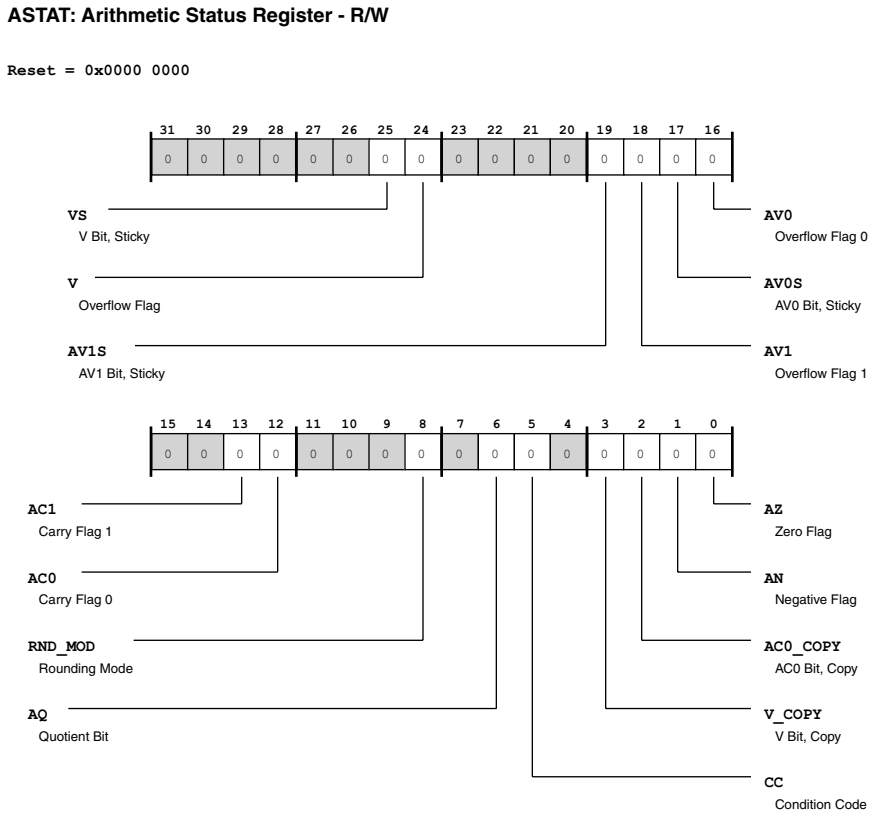


Figure 4-10: ASTAT Register Diagram

Table 4-19: ASTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
25 (R/W)	VS	V Bit, Sticky. The ASTAT.VS bit is set if ASTAT.VS is set; unaffected otherwise.
24 (R/W)	V	Overflow Flag. The ASTAT.V bit is set if the most recent ALU0 or MAC0 operation result overflows; cleared if operation generates no overflow,
19 (R/W)	AV1S	AV1 Bit, Sticky. The ASTAT.AV1S bit is set if ASTAT.AV1 is set; unaffected otherwise.
18 (R/W)	AV1	Overflow Flag 1. The ASTAT.AV1 bit is set if the most recent MAC1 operation result placed in A1 overflows; cleared if result placed in A1 generates no overflow; sticky for MAC.
17 (R/W)	AV0S	AV0 Bit, Sticky. The ASTAT.AV0S bit is set if ASTAT.AV0 is set; unaffected otherwise.
16 (R/W)	AV0	Overflow Flag 0. The ASTAT.AV0 bit is set if the most recent ALU0 or MAC0 operation result placed in A0 overflows; cleared if result placed in A0 generates no overflow; sticky for MAC.

Table 4-19: ASTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
13 (R/W)	AC1	Carry Flag 1. The ASTAT.AC1 bit is set if the most recent ALU1 operation using A1 generates a carry; cleared if operation using A1 generates no carry.	
12 (R/W)	AC0	Carry Flag 0. The ASTAT.AC0 bit is set if the ALU0 operation using A0 generates a carry; cleared if operation using A0 generates no carry.	
8 (R/W)	RND_MOD	Rounding Mode. The ASTAT.RND_MOD bit is not affected by result status. Instead, this bit is used to select the rounding mode for arithmetic instructions that support rounding.	
		0	Unbiased rounding
		1	Biased rounding
6 (R/W)	AQ	Quotient Bit. (Divs and Divq instructions only) The ASTAT.AQ bit equals the dividend MSB exclusive-OR divisor MSB, where the dividend is a 32-bit value and divisor is a 16-bit value.	
5 (R/W)	CC	Condition Code. (TestSet instruction only) The ASTAT.CC bit is set if the addressed value is zero; cleared if the addressed value is non-zero 0x0.	
3 (R/W)	V_COPY	V Bit, Copy. The ASTAT.V_COPY bit is set if ASTAT.V_COPY is set; cleared if ASTAT.V_COPY is cleared.	
2 (R/W)	AC0_COPY	AC0 Bit, Copy. The ASTAT.AC0_COPY bit is set if ASTAT.AC0 is set; cleared if ASTAT.AC0 is cleared.	
1 (R/W)	AN	Negative Flag. The ASTAT.AN bit is set if the most recent ALU0 or shifter operation result is negative; cleared if result is non-negative.	
0 (R/W)	AZ	Zero Flag. The ASTAT.AZ bit is set if the most recent ALU0 or shifter operation result is zero; cleared if result is non-zero.	

Return from Subroutine Register

The CALL instruction calls a subroutine from an address that a pointer register points to or by using a PC-relative offset. After the CALL instruction executes, the RETS register contains the address of the next instruction. The RETS register holds the address (of the next instruction) for the return from subroutine (RTS) instruction. The RETS register is not a memory mapped register, but this register is directly readable and writable by move instructions. Also, this register may be pushed to or popped from the system stack. Note that load/store operations and immediate load operations are not supported for the RETS register.

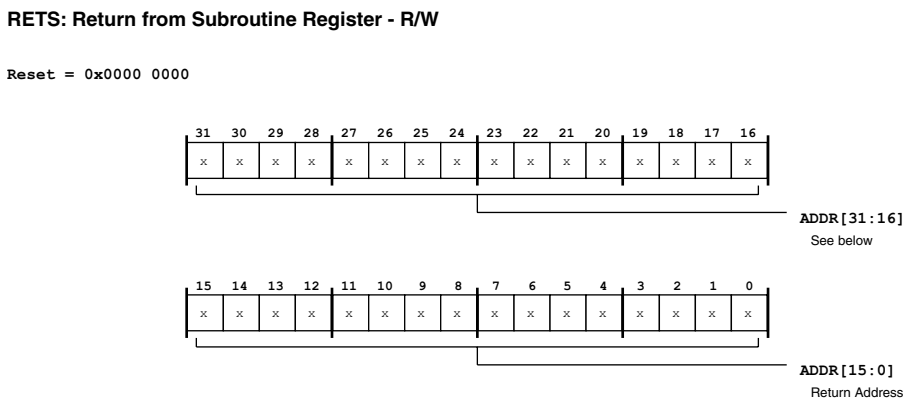


Figure 4-11: RETS Register Diagram

Table 4-20: RETS Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Return Address. The RETS.ADDR bits hold the address value for the RTS instruction.

Loop Count Register

The LTn register holds the address of the first instruction within a loop. The LBn register holds the address of the last instruction of the loop. The LCn maintains a count of the remaining iterations of the loop. These loop registers implement zero-overhead loops, using hardware counters instead of software instructions to evaluate loop conditions. The loop setup (LSETUP) instruction initializes the LTn, LBn, and LCn registers.

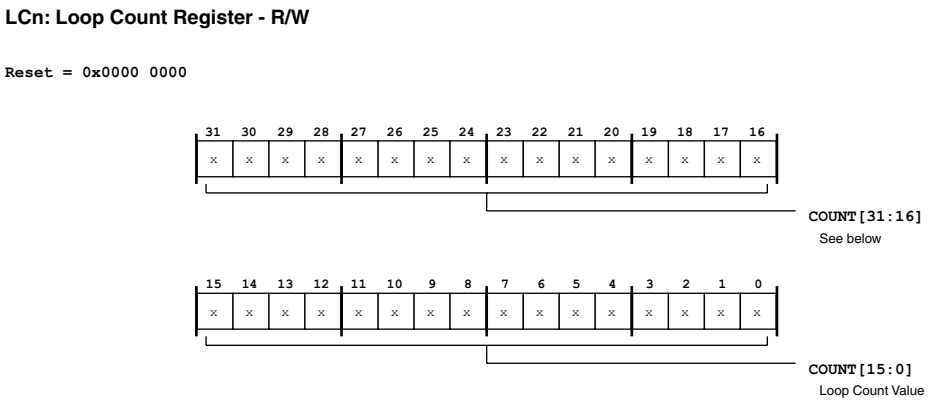


Figure 4-12: LCn Register Diagram

Table 4-21: LCn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	COUNT	Loop Count Value. The LCn.COUNT bits hold the count value of the remaining iterations of the loop.

Loop Top Register

The LTn register holds the address of the first instruction within a loop. The LBn register holds the address of the last instruction of the loop. The LCn maintains a count of the remaining iterations of the loop. These loop registers implement zero-overhead loops, using hardware counters instead of software instructions to evaluate loop conditions. The loop setup (LSETUP) instruction initializes the LTn, LBn, and LCn registers.

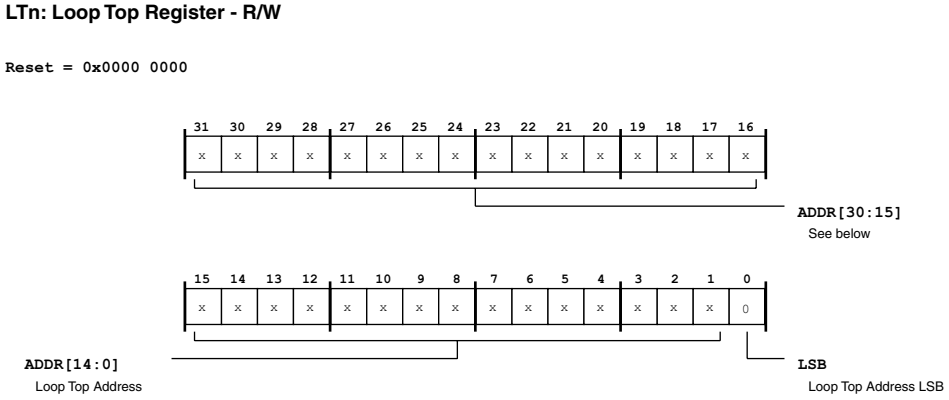


Figure 4-13: LTn Register Diagram

Table 4-22: LTn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:1 (R/W)	ADDR	Loop Top Address. The LTn.ADDR bits hold the address of the first instruction in the loop.
0 (R0/NW)	LSB	Loop Top Address LSB. The LTn.LSB bit is the LSB of address of the first instruction in the loop. (Always =1.)

Loop Bottom Register

The LTn register holds the address of the first instruction within a loop. The LBn register holds the address of the last instruction of the loop. The LCn maintains a count of the remaining iterations of the loop. These loop registers implement zero-overhead loops, using hardware counters instead of software instructions

to evaluate loop conditions. The loop setup (LSETUP) instruction initializes the LTn, LBn, and LCn registers.

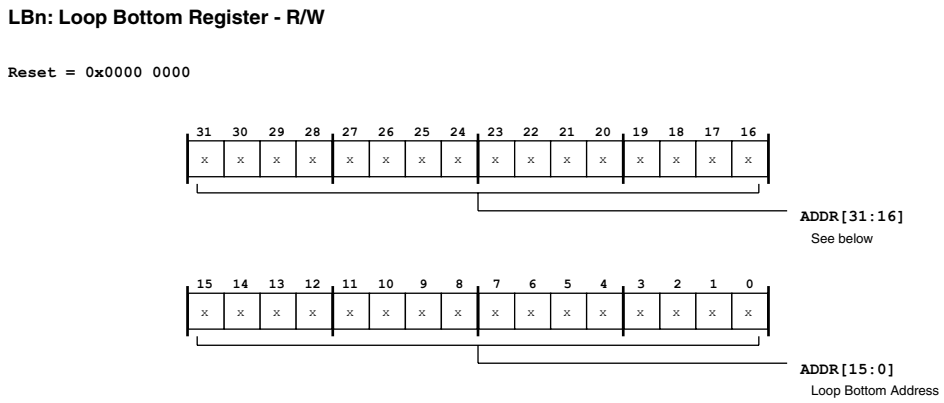


Figure 4-14: LBn Register Diagram

Table 4-23: LBn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Loop Bottom Address. The LBn.ADDR bits hold the address of the last instruction in the loop

Cycle Count (32 LSBs) Register

The CYCLES register holds the 32 least significant bits of the cycle count. The counter is enabled using the SYSCFG.CCEN bit. For more information, see the SYSCFG.CCEN bit description.

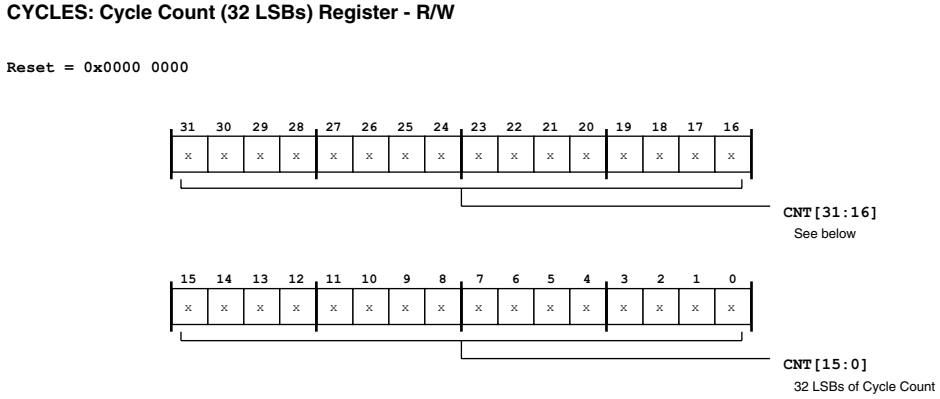


Figure 4-15: CYCLES Register Diagram

Table 4-24: CYCLES Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	32 LSBs of Cycle Count. The CYCLES.CNT bits hold the least significant 32 bits of the cycle count value.

Cycle Count (32 MSBs) Register

The CYCLES2 register holds the 32 most significant bits of the cycle count. The counter is enabled using the SYSCFG.CCEN bit. For more information, see the SYSCFG.CCEN bit description.

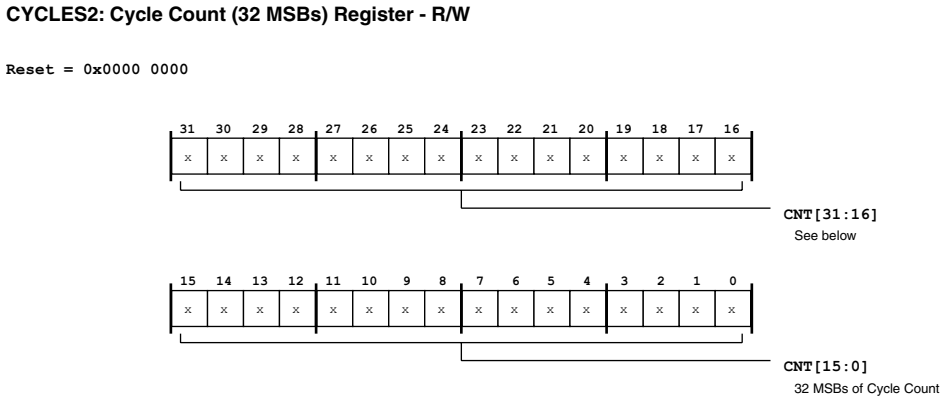


Figure 4-16: CYCLES2 Register Diagram

Table 4-25: CYCLES2 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	32 MSBs of Cycle Count. The CYCLES2.CNT bits hold the most significant 32 bits of the cycle count value.

User Stack Pointer Register

In many respects, the stack and frame pointer registers perform like the other Pn registers. The SP and FP registers may act as general pointers in any of the load/store instructions. But, the SP and FP registers have additional functionality. To speed up context switching, there are two stack pointer registers, a user stack pointer (USP) and a supervisor stack pointer (SP). In assembly code syntax, one should always just use the SP syntax. Based on the current processor mode (supervisor or user), the correct register stack pointer will be used. For context switching, in supervisor mode, one may explicitly refer to the user stack pointer by using the USP syntax. Trying to use USP syntax in user mode causes an exception. Some load and store instructions use SP and FP implicitly. These instructions include LINK and UNLINK instructions, which control stack frame space and manage the SP register.

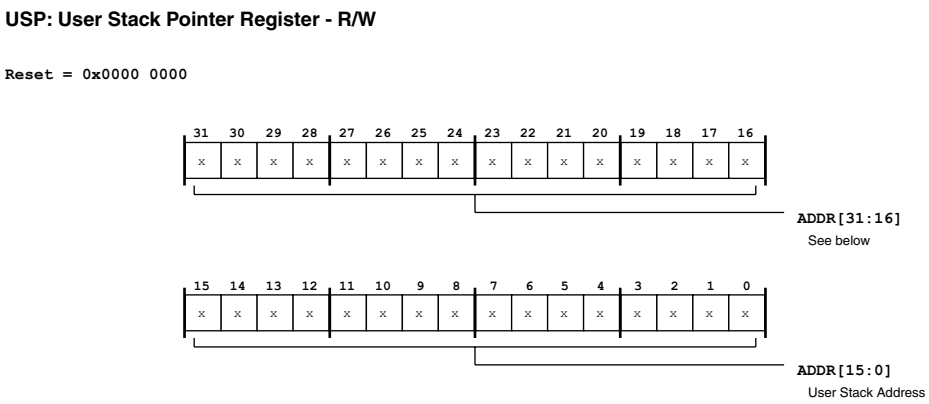


Figure 4-17: USP Register Diagram

Table 4-26: USP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	User Stack Address. The USP.ADDR bit field (bits 0:31) holds an address (pointer) for address calculations or stack operations in user mode.

Sequencer Status Register

The SEQSTAT register contains information about the current state of the sequencer and diagnostic information from the most recent event. This register is accessible only in supervisor mode.

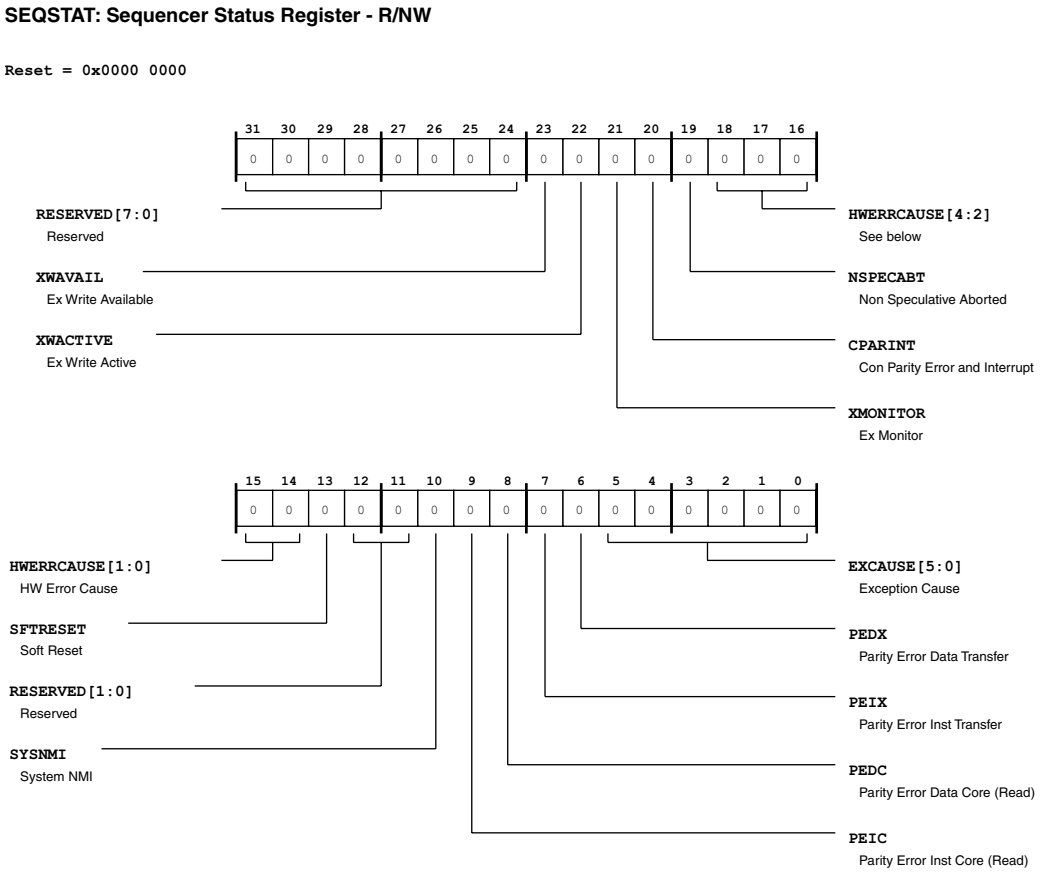


Figure 4-18: SEQSTAT Register Diagram

Table 4-27: SEQSTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:24 (R0/NW)	RESERVED	Reserved.	
23 (R/NW)	XWAVAIL	Ex Write Available. The SEQSTAT.XWAVAIL bit indicates whether an exclusive write response is available.	
		0	No status
		1	Write response available
22 (R/NW)	XWACTIVE	Ex Write Active. The SEQSTAT.XWACTIVE bit indicates whether an exclusive write is currently active.	
		0	No status
		1	Write currently active

Table 4-27: SEQSTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
21 (R/NW)	XMONITOR	Ex Monitor. The SEQSTAT.XMONITOR bit indicates monitors for exclusive or open access.	
		0	Open access
		1	Exclusive access
20 (R/W1C)	CPARINT	Con Parity Error and Interrupt. The SEQSTAT.CPARINT bit indicates status for a concurrent parity error and interrupt. If set, this bit indicates that a parity error could not suppress a concurrent interrupt. If the concurrent interrupt was an NMI or had higher priority, the parity error may have occurred on the return address of the current interrupt service routine. If the concurrent interrupt had lower priority than an NMI, the parity error may have occurred on the return address of previous interrupt service routine.	
		0	No status
		1	Error and interrupt occurred
19 (R/W1C)	NSPECABT	Non Speculative Aborted. The SEQSTAT.NSPECABT bit indicates whether a non-speculative access was aborted. If set, this bit indicates that a non-speculative access was interrupted by an NMI or was interrupted by a hardware emulator interrupts. If the access was to a resource that has read side effects (such as, a FIFO), some information (data from the access) may have been lost.	
		0	No status
		1	Access aborted
18:14 (R/NW)	HWERRCAUSE	HW Error Cause. The SEQSTAT.HWERRCAUSE bits hold a value indicated the cause of the last hardware error generated by the processor core.	
		18	Overflowed performance monitor count
		24	Executed RAISE 5 instruction
13 (R/NW)	SFTRESET	Soft Reset. The SEQSTAT.SFTRESET bits indicates whether the most recent processor reset was a software reset.	
		0	Not software reset (most recent reset)
		1	Software reset occurred (most recent reset)
12:11 (R0/NW)	RESERVED	Reserved.	
10 (R/W1C)	SYSNMI	System NMI. The SEQSTAT.SYSNMI bit indicates whether the system NMI input is active.	
		0	No status
		1	NMI active

Table 4-27: SEQSTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
9 (R/NW)	PEIC	Parity Error Inst Core (Read). The SEQSTAT.PEIC bit indicates a parity error occurred on an L1 instruction memory read by the processor.	
		0	No status
		1	Parity error occurred
8 (R/NW)	PEDC	Parity Error Data Core (Read). The SEQSTAT.PEDC bit indicates a parity error occurred on an L1 data memory read by the processor.	
		0	No status
		1	Parity error occurred
7 (R/NW)	PEIX	Parity Error Inst Transfer. The SEQSTAT.PEIX bit indicates a parity error occurred on an L1 instruction memory read by the system, such as a DMA transfer out of L1 instruction memory.	
		0	No status
		1	Parity error occurred
6 (R/NW)	PEDX	Parity Error Data Transfer. The SEQSTAT.PEDX bit indicates a parity error occurred on an L1 data memory read by the system, such as a cache write back or a DMA transfer out of L1 data memory.	
		0	No status
		1	Parity error occurred

Table 4-27: SEQSTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
5:0 (R/NW)	EXCAUSE	Exception Cause. The SEQSTAT.EXCAUSE bits hold a value, which indicates the cause of the most recently executed exception.	
		00xxxx	EXCPT instruction
		16	Single step exception
		17	Emulator trace buffer overflow
		33	Undefined instruction
		34	Illegal combination of instructions
		35	DAG protection violation
		36	DAG misaligned access
		37	Unrecoverable event
		38	DAG CPLB miss
		39	DAG multiple CPLB hits
		40	Emulator watchpoint match
		42	I-fetch misaligned access
		43	I-fetch protection violation
		44	I-fetch CPLB miss
		45	I-fetch multiple CPLB hits
		46	Protection violation, illegal use of supervisor resource

System Configuration Register

The SYSCFG register controls the configuration of the processor. This register is accessible only from the supervisor mode.

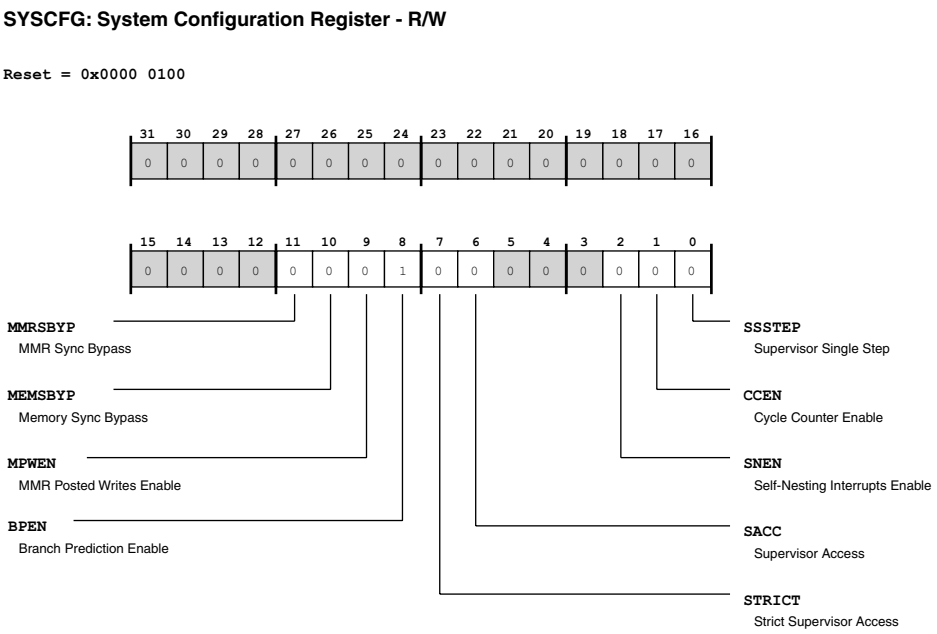


Figure 4-19: SYSCFG Register Diagram

Table 4-28: SYSCFG Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
11 (R/W)	MMRSBYP	MMR Sync Bypass. The SYSCFG.MMRSBYP bit enables bypass mode for clock domain synchronization in memory mapped register interface. Enabling this feature reduces read latency.	
		0	No bypass
		1	Bypass
10 (R/W)	MEMSBYP	Memory Sync Bypass. The SYSCFG.MEMSBYP bit enables bypass mode for clock domain synchronization in system memory bus interface. Enabling this feature reduces read latency.	
		0	No bypass
		1	Bypass
9 (R/W)	MPWEN	MMR Posted Writes Enable. The SYSCFG.MPWEN bit enables support for posting consecutive system MMR writes to the system fabric. When disabled, the processor waits for each response before allowing a new write to go into the system. Immediately after changing this mode (either enabling or disabling), an SSYNC instruction must be executed to allow the system to finish any outstanding writes before changing the rules on how writes go out to the system.	
		0	Disable
		1	Enable

Table 4-28: SYSCFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
8 (R/W)	BPEN	Branch Prediction Enable. The SYSCFG.BPEN bit selects whether the program sequencer uses dynamic or static branch prediction operation. For more information, see the Branch Prediction section of the Program Sequencer chapter.	
		0	Use static prediction
		1	Use dynamic prediction
7 (R/W)	STRICT	Strict Supervisor Access. The SYSCFG.STRICT bit restricts additional resources (beyond those restricted on the ADSP-BF5xx/BF6xx Blackfin processors) to require supervisor mode access. When enabled, accessing any of these additional resources in user mode (for example, executing an IDLE instruction) causes an illegal supervisor access exception. For more information, see the Protected Resources and Instructions section in the Operating Modes and States chapter.	
		0	Disable (normal/previous Blackfin access operation)
		1	Enable (strict supervisor access operation)
6 (R/W)	SACC	Supervisor Access. The SYSCFG.SACC bit selects whether supervisor mode access is permitted when the processor is not servicing any events or whether supervisor mode only is permitted when the processor is servicing an event.	
		0	Disable (access only when servicing an event)
		1	Enable (access whether or not servicing any events)
2 (R/W)	SNEN	Self-Nesting Interrupts Enable. The SYSCFG.SNEN bit enables self-nesting interrupt operation. Unlike interrupts during normal operation, interrupts during self-nesting may be interrupted by events at the same priority level. Self-nesting is supported for any interrupt level generated with the RAISE instruction, as well as for core level interrupts.	
		0	Disable (normal interrupt operation)
		1	Enable (self-nesting interrupt operation)
1 (R/W)	CCEN	Cycle Counter Enable. The SYSCFG.CCEN bit enables cycle counter operation. When the cycle counter is enabled, it counts core clock (CCLK) cycles, incrementing with each cycle. All cycles are counted (including wait states) in both user mode and supervisor modes. The cycle counter stops counting while the processor is in emulator mode. The cycle counter is 64 bits wide, and the count is stored in the CYCLES and CYCLES2 registers. The least significant 32 bits are stored in CYCLES.	
		0	Disable cycle counter
		1	Enable cycle counter

Table 4-28: SYSCFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
0 (R/W)	SSSTEP	Supervisor Single Step. The SYSCFG.SSSTEP bit enables single step operation, in which a supervisor exception occurs after the processor executes each instruction. This bit only applies to executing instructions in user mode or to processing interrupts in supervisor mode. The SYSCFG.SSSTEP bit is ignored if the core is processing an exception or higher-priority event. If precise exception timing is required, a CSYNC instruction must be used after setting this bit.	
		0	Disable (normal operation)
		1	Enable (single step operation)

Return from Interrupt Register

When the processor vectors to an interrupt to handle an event, the processor saves the address to which program flow returns after the interrupt is serviced. The RETI register holds the address for the return from interrupt (RTI) instruction. The RETI register is not a memory mapped register, but this register is directly readable and writable by move instructions. Also, this register may be pushed to or popped from the system stack. Note that load/store operations and immediate load operations are not supported for the RETI register.

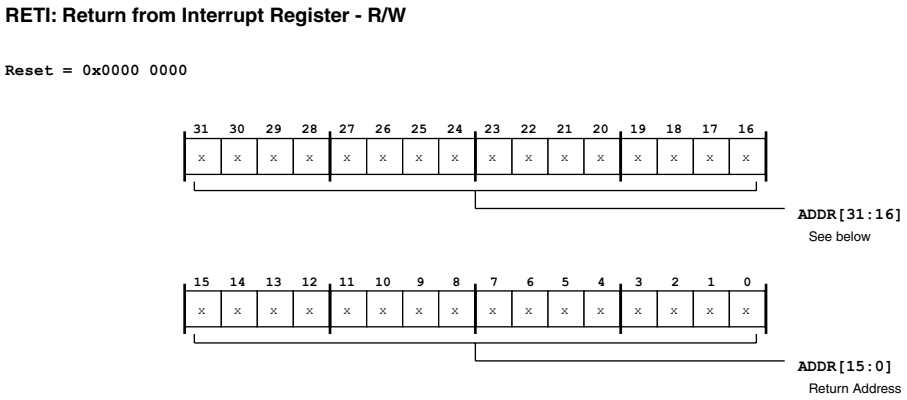


Figure 4-20: RETI Register Diagram

Table 4-29: RETI Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Return Address. The RETI.ADDR bits hold the address value for the RTI instruction.

Return from Exception Register

When the processor vectors to an interrupt to handle an event, the processor saves the address to which program flow returns after the interrupt is serviced. The RETX register holds the address for the return from exception interrupt (RTX) instruction. The RETX register is not a memory mapped register, but this register is directly readable and writable by move instructions. Also, this register may be pushed to or popped from the system stack. Note that load/store operations and immediate load operations are not supported for the RETX register.

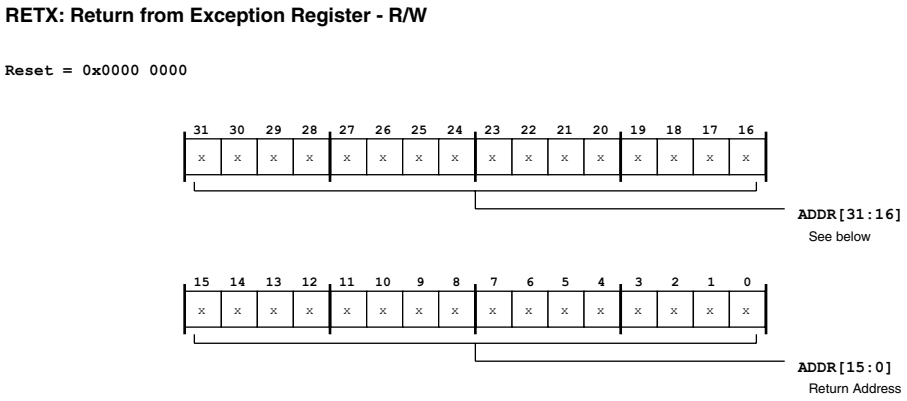


Figure 4-21: RETX Register Diagram

Table 4-30: RETX Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Return Address. The RETX.ADDR bits hold the address value for the RTX instruction.

Return from NMI Register

When the processor vectors to an interrupt to handle an event, the processor saves the address to which program flow returns after the interrupt is serviced. The RETN register holds the address for the return from NMI (RTN) instruction. The RETN register is not a memory mapped register, but this register is directly readable and writable by move instructions. Also, this register may be pushed to or popped from the system stack. Note that load/store operations and immediate load operations are not supported for the RETN register.

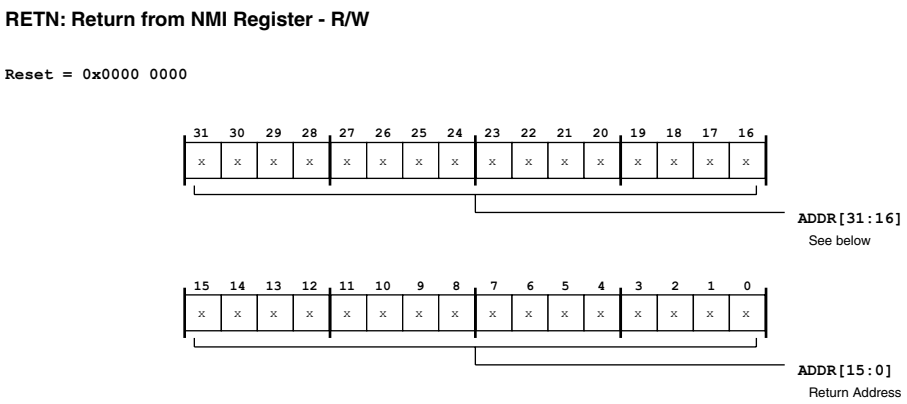


Figure 4-22: RETN Register Diagram

Table 4-31: RETN Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Return Address. The RETN.ADDR bits hold the address value for the RTN instruction.

Return from Emulator Register

When the processor vectors to an interrupt to handle an event, the processor saves the address to which program flow returns after the interrupt is serviced. The RETE register holds the address for the return from emulator interrupt (RTE) instruction. The RETE register is not a memory mapped register, but this register is directly readable and writable by move instructions. Also, this register may be pushed to or popped from the system stack. Note that load/store operations and immediate load operations are not supported for the RETE register.

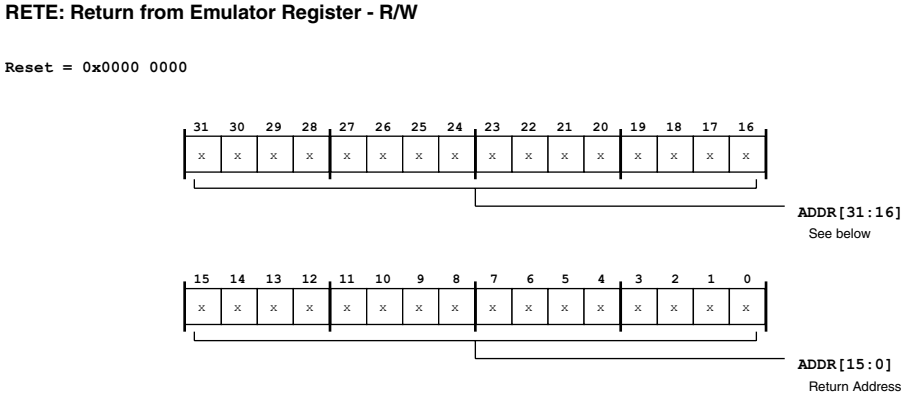


Figure 4-23: RETE Register Diagram

Table 4-32: RETE Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Return Address. The RETE.ADDR bits hold the address value for the RTE instruction.

ADSP-BF70x ICU Register Descriptions

Interrupt Control Unit (ICU) contains the following registers.

Table 4-33: ADSP-BF70x ICU Register List

Name	Description
EVTn	Event Vector Table Registers
EVT_OVERRIDE	Event Vector Table Override Register
IMASK	Interrupt Mask Register
IPEND	Interrupt Pending Register
ILAT	Interrupt Latch Register
ICU_CID	Context ID Register
CEC_SID	System ID Register

Event Vector Table Registers

The EVTn registers provide a table of interrupt service routine addresses, providing an entry for each possible core event. Entries are accessed as memory mapped registers, and each entry can be programmed at reset with the vector address for the corresponding interrupt service routine. When an event occurs, instruction fetch starts at the address location in the EVTn entry for that event.

The processor architecture allows unique addresses to be programmed into each of the interrupt vectors; that is, interrupt vectors are not determined by a fixed offset from an interrupt vector table base address. This approach minimizes latency by not requiring a long jump from the vector table to the actual ISR code.

See the IMASK, IPEND, or ILAT register descriptions for a list of interrupt vectors.

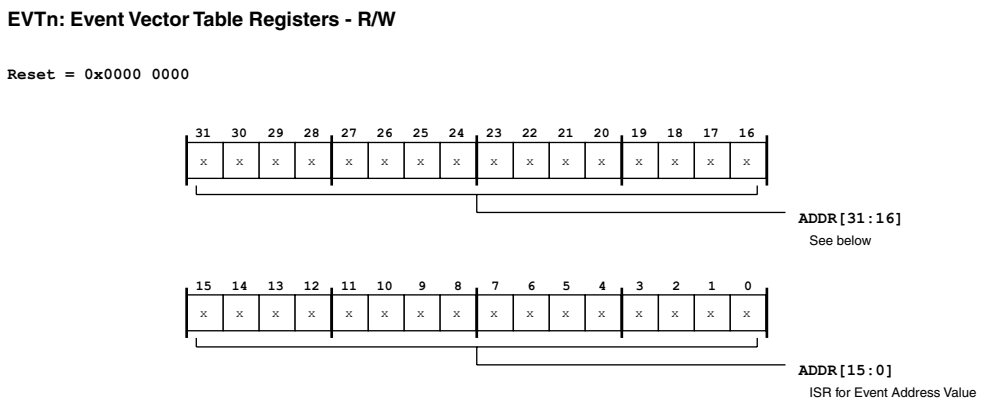


Figure 4-24: EVTn Register Diagram

Table 4-34: EVTn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	ISR for Event Address Value. The EVTn.ADDR bits hold the interrupt service routine address value for the corresponding event table entry.

Event Vector Table Override Register

The EVT_OVERRIDE register determines whether one or more of the processor interrupt vectors takes its vector address from the address on the external bus at reset. For example, if EVT_OVERRIDE.IVG7 is set (=1), the IVG7 interrupt vectors to the address in the EVT1 register (EVT entry for the reset event), rather than vectoring to the address in EVT7 register (the address in the IVG7 entry of the EVT is ignored.) The EVT entry for the reset event is the address present on the external bus at reset (while reset is asserted). The reset interrupt always vectors to the address on that bus, without exception.

Note: The rationale for this feature is that it eliminates the need for double indirection to vector an interrupt to an externally supplied address.

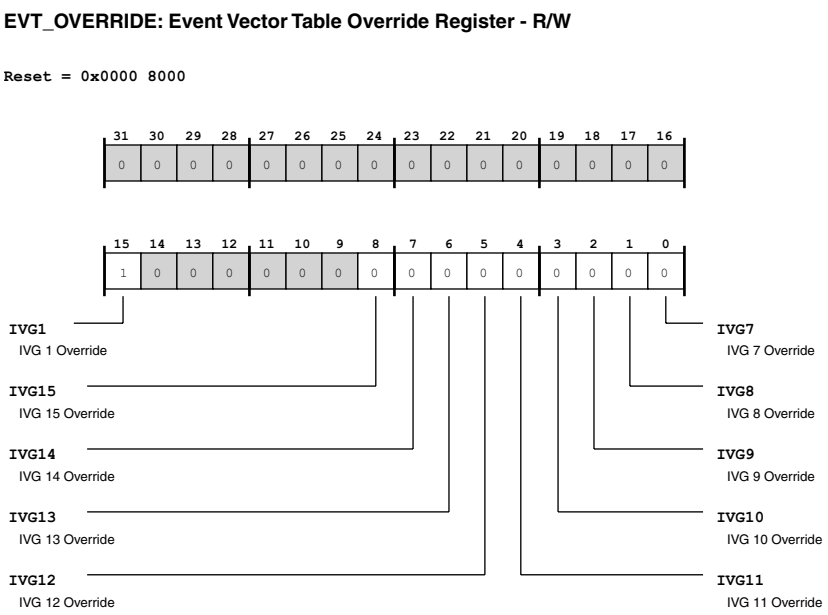


Figure 4-25: EVT_OVERRIDE Register Diagram

Table 4-35: EVT_OVERRIDE Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15 (R/W)	IVG1	IVG 1 Override. The EVT_OVERRIDE.IVG1 bit overrides (if set, =1) the corresponding EVTn address for this interrupt, directing the processor to take the vector address EVT1 register.
8 (R/W)	IVG15	IVG 15 Override. The EVT_OVERRIDE.IVG15 bit overrides (if set, =1) the corresponding EVTn address for this interrupt, directing the processor to take the vector address EVT1 register.
7 (R/W)	IVG14	IVG 14 Override. The EVT_OVERRIDE.IVG14 bit overrides (if set, =1) the corresponding EVTn address for this interrupt, directing the processor to take the vector address EVT1 register.
6 (R/W)	IVG13	IVG 13 Override. The EVT_OVERRIDE.IVG13 bit overrides (if set, =1) the corresponding EVTn address for this interrupt, directing the processor to take the vector address EVT1 register.
5 (R/W)	IVG12	IVG 12 Override. The EVT_OVERRIDE.IVG12 bit overrides (if set, =1) the corresponding EVTn address for this interrupt, directing the processor to take the vector address EVT1 register.

Table 4-35: EVT_OVERRIDE Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
4 (R/W)	IVG11	IVG 11 Override. The EVT_OVERRIDE.IVG11 bit overrides (if set, =1) the corresponding EVTn address for this interrupt, directing the processor to take the vector address EVT1 register.
3 (R/W)	IVG10	IVG 10 Override. The EVT_OVERRIDE.IVG10 bit overrides (if set, =1) the corresponding EVTn address for this interrupt, directing the processor to take the vector address EVT1 register.
2 (R/W)	IVG9	IVG 9 Override. The EVT_OVERRIDE.IVG9 bit overrides (if set, =1) the corresponding EVTn address for this interrupt, directing the processor to take the vector address EVT1 register.
1 (R/W)	IVG8	IVG 8 Override. The EVT_OVERRIDE.IVG8 bit overrides (if set, =1) the corresponding EVTn address for this interrupt, directing the processor to take the vector address EVT1 register.
0 (R/W)	IVG7	IVG 7 Override. The EVT_OVERRIDE.IVG7 bit overrides (if set, =1) the corresponding EVTn address for this interrupt, directing the processor to take the vector address EVT1 register.

Interrupt Mask Register

The IMASK register indicates which interrupt levels are allowed to be taken. This register may be read and written in supervisor mode. Bits 15-5 permit unmasking (enabling) or masking (disabling) interrupts with these priority levels; bits 4-0 are hard-coded to =1, and events of these priority levels are always unmasked (enabled). If IMASK[n] == 1 and ILAT[n] == 1, interrupt n is taken if a higher priority is not already recognized. If IMASK[n] == 0 and ILAT[n] is set by interrupt n, interrupt n will not be taken, and ILAT[n] remains set.

IMASK: Interrupt Mask Register - R/W

Reset = 0x0000 001f

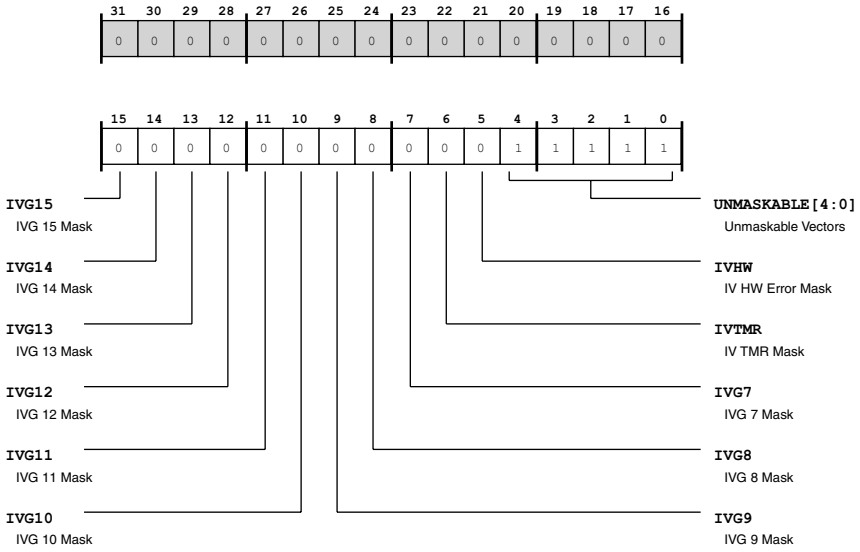


Figure 4-26: IMASK Register Diagram

Table 4-36: IMASK Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15 (R/W)	IVG15	IVG 15 Mask. The IMASK.IVG15 bit unmasks (enables, =1) or masks (disables, =0) interrupts with this priority level.
14 (R/W)	IVG14	IVG 14 Mask. The IMASK.IVG14 bit unmasks (enables, =1) or masks (disables, =0) interrupts with this priority level.
13 (R/W)	IVG13	IVG 13 Mask. The IMASK.IVG13 bit unmasks (enables, =1) or masks (disables, =0) interrupts with this priority level.
12 (R/W)	IVG12	IVG 12 Mask. The IMASK.IVG12 bit unmasks (enables, =1) or masks (disables, =0) interrupts with this priority level.
11 (R/W)	IVG11	IVG 11 Mask. The IMASK.IVG11 bit unmasks (enables, =1) or masks (disables, =0) interrupts with this priority level.
10 (R/W)	IVG10	IVG 10 Mask. The IMASK.IVG10 bit unmasks (enables, =1) or masks (disables, =0) interrupts with this priority level.
9 (R/W)	IVG9	IVG 9 Mask. The IMASK.IVG9 bit unmasks (enables, =1) or masks (disables, =0) interrupts with this priority level.

Table 4-36: IMASK Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
8 (R/W)	IVG8	IVG 8 Mask. The IMASK.IVG8 bit unmasks (enables, =1) or masks (disables, =0) interrupts with this priority level.
7 (R/W)	IVG7	IVG 7 Mask. The IMASK.IVG7 bit unmasks (enables, =1) or masks (disables, =0) interrupts with this priority level.
6 (R/W)	IVTMR	IV TMR Mask. The IMASK.IVTMR bit unmasks (enables, =1) or masks (disables, =0) the timer interrupts.
5 (R/W)	IVHW	IV HW Error Mask. The IMASK.IVHW bit unmasks (enables, =1) or masks (disables, =0) the hardware error interrupt.
4:0 (R/NW)	UNMASKABLE	Unmaskable Vectors. The IMASK.UNMASKABLE bits are always unmasked (enabled, =1), enabling all interrupts with these priority levels.

Interrupt Pending Register

The IPEND register keeps track of all currently nested interrupts. Each bit in this register indicates that the corresponding interrupt is currently active or nested at some level. The IPEND register may be read in supervisor mode, but may not be written. The IPEND[4] bit is used by the event controller to temporarily disable interrupts on entry and exit to an interrupt service routine.

When an event is processed, the corresponding bit in IPEND is set. The least significant bit in this register that is currently set indicates the interrupt that is currently being serviced. At any given time, this register holds the current status of all nested events.

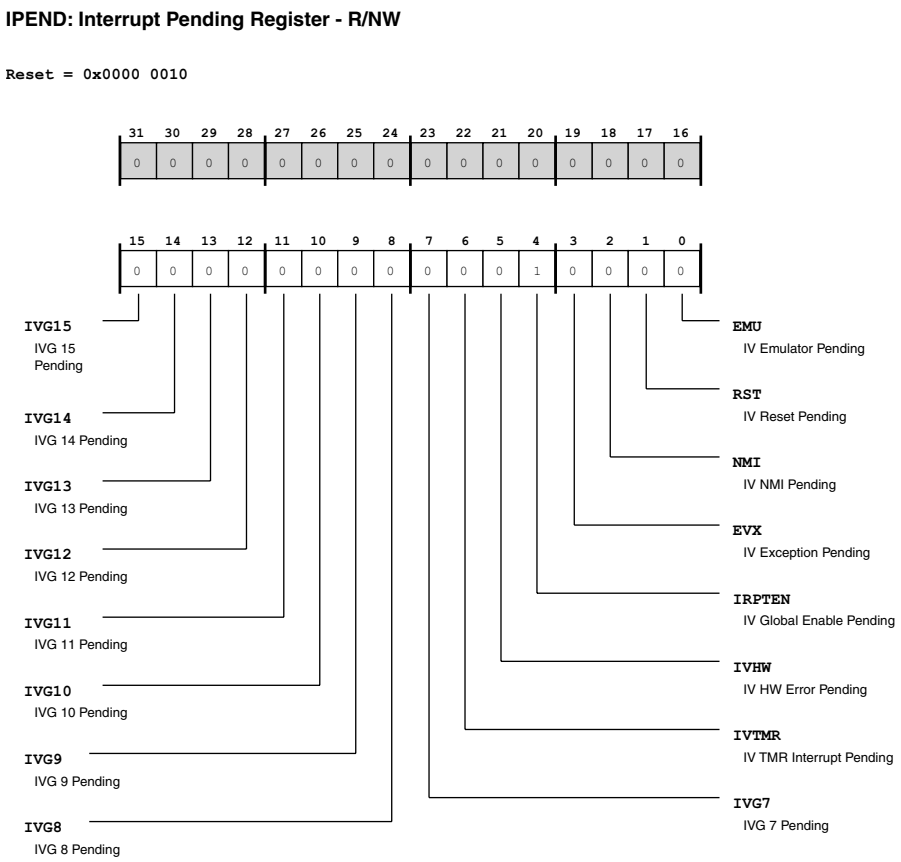


Figure 4-27: IPEND Register Diagram

Table 4-37: IPEND Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15 (R/NW)	IVG15	IVG 15 Pending. The IPEND.IVG15 bit if set (=1) indicates that corresponding interrupt is currently active or nested at some level.
14 (R/NW)	IVG14	IVG 14 Pending. The IPEND.IVG14 bit if set (=1) indicates that corresponding interrupt is currently active or nested at some level.
13 (R/NW)	IVG13	IVG 13 Pending. The IPEND.IVG13 bit if set (=1) indicates that corresponding interrupt is currently active or nested at some level.
12 (R/NW)	IVG12	IVG 12 Pending. The IPEND.IVG12 bit if set (=1) indicates that corresponding interrupt is currently active or nested at some level.
11 (R/NW)	IVG11	IVG 11 Pending. The IPEND.IVG11 bit if set (=1) indicates that corresponding interrupt is currently active or nested at some level.

Table 4-37: IPEND Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
10 (R/NW)	IVG10	IVG 10 Pending. The IPEND.IVG10 bit if set (=1) indicates that corresponding interrupt is currently active or nested at some level.
9 (R/NW)	IVG9	IVG 9 Pending. The IPEND.IVG9 bit if set (=1) indicates that corresponding interrupt is currently active or nested at some level.
8 (R/NW)	IVG8	IVG 8 Pending. The IPEND.IVG8 bit if set (=1) indicates that corresponding interrupt is currently active or nested at some level.
7 (R/NW)	IVG7	IVG 7 Pending. The IPEND.IVG7 bit if set (=1) indicates that corresponding interrupt is currently active or nested at some level.
6 (R/NW)	IVTMR	IV TMR Interrupt Pending. The IPEND.IVTMR bit if set (=1) indicates that timer interrupt is currently active or nested at some level.
5 (R/NW)	IVHW	IV HW Error Pending. The IPEND.IVHW bit if set (=1) indicates that hardware error interrupt is currently active or nested at some level.
4 (R/NW)	IRPTEN	IV Global Enable Pending. The IPEND.IRPTEN bit if set (=1) indicates that global interrupt enable is currently active or nested at some level.
3 (R/NW)	EVX	IV Exception Pending. The IPEND.EVX bit if set (=1) indicates that exception interrupt is currently active or nested at some level.
2 (R/NW)	NMI	IV NMI Pending. The IPEND.NMI bit if set (=1) indicates that non-maskable interrupt is currently active.
1 (R/NW)	RST	IV Reset Pending. The IPEND.RST bit if set (=1) indicates that reset interrupt is currently active.
0 (R/NW)	EMU	IV Emulator Pending. The IPEND.EMU bit if set (=1) indicates that emulator interrupt is currently active.

Interrupt Latch Register

Each bit in the ILAT register indicates that the corresponding event is latched, but not yet accepted into the processor. The bit is reset before the first instruction in the corresponding interrupt service routing is executed. At the point the interrupt is accepted, ILAT[n] is cleared, and IPEND[n] is set simultaneously. The ILAT register may be read in supervisor mode. Writes to this register are used to clear bits only (in supervisor mode). To clear bit n from ILAT, first make sure that IMASK[n] == 0, then write ILAT[n] =1. This write functionality to ILAT is provided for cases where latched interrupt requests need to be cleared (canceled) instead of serviced.

The RAISE instruction can be used to set ILAT[15] through ILAT[5], and also ILAT[2] or ILAT[1]. Only the JTAG TRST pin can clear ILAT[0].

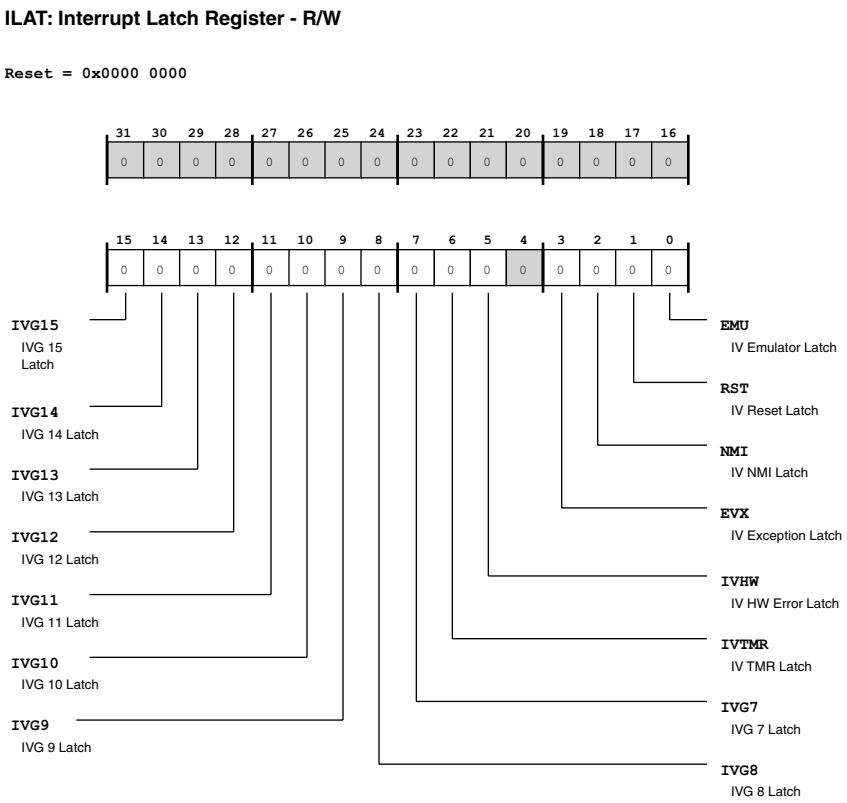


Figure 4-28: ILAT Register Diagram

Table 4-38: ILAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15 (R/W1C)	IVG15	IVG 15 Latch. The ILAT.IVG15 bit if set (=1) indicates that the corresponding event is latched--but not yet accepted--into the processor.
14 (R/W1C)	IVG14	IVG 14 Latch. The ILAT.IVG14 bit if set (=1) indicates that the corresponding event is latched--but not yet accepted--into the processor.
13 (R/W1C)	IVG13	IVG 13 Latch. The ILAT.IVG13 bit if set (=1) indicates that the corresponding event is latched--but not yet accepted--into the processor.
12 (R/W1C)	IVG12	IVG 12 Latch. The ILAT.IVG12 bit if set (=1) indicates that the corresponding event is latched--but not yet accepted--into the processor.

Table 4-38: ILAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
11 (R/W1C)	IVG11	IVG 11 Latch. The ILAT.IVG11 bit if set (=1) indicates that the corresponding event is latched--but not yet accepted--into the processor.
10 (R/W1C)	IVG10	IVG 10 Latch. The ILAT.IVG10 bit if set (=1) indicates that the corresponding event is latched--but not yet accepted--into the processor.
9 (R/W1C)	IVG9	IVG 9 Latch. The ILAT.IVG9 bit if set (=1) indicates that the corresponding event is latched--but not yet accepted--into the processor.
8 (R/W1C)	IVG8	IVG 8 Latch. The ILAT.IVG8 bit if set (=1) indicates that the corresponding event is latched--but not yet accepted--into the processor.
7 (R/W1C)	IVG7	IVG 7 Latch. The ILAT.IVG7 bit if set (=1) indicates that the corresponding event is latched--but not yet accepted--into the processor.
6 (R/W1C)	IVTMR	IV TMR Latch. The ILAT.IVTMR bit if set (=1) indicates that the timer interrupt is latched--but not yet accepted--into the processor.
5 (R/W1C)	IVHW	IV HW Error Latch. The ILAT.IVHW bit if set (=1) indicates that the hardware error interrupt is latched--but not yet accepted--into the processor.
3 (R/NW)	EVX	IV Exception Latch. The ILAT.EVX bit if set (=1) indicates that the exception interrupt is latched--but not yet accepted--into the processor.
2 (R/NW)	NMI	IV NMI Latch. The ILAT.NMI bit if set (=1) indicates that the non-maskable interrupt is latched--but not yet accepted--into the processor.
1 (R/NW)	RST	IV Reset Latch. The ILAT.RST bit if set (=1) indicates that the reset interrupt is latched--but not yet accepted--into the processor.
0 (R/NW)	EMU	IV Emulator Latch. The ILAT.EMU bit if set (=1) indicates that the emulator interrupt is latched--but not yet accepted--into the processor.

Context ID Register

The ICU_CID register holds a 32-bit context ID that is specified by software. This register is defined as part of the debug trace solution. The software defined context id is captured by the program flow trace block for comparison (trace filtering) and may be included in the trace packets.

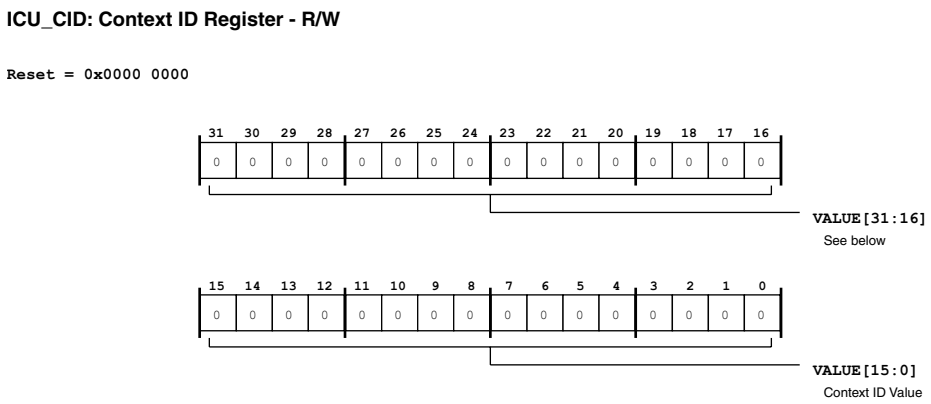


Figure 4-29: ICU_CID Register Diagram

Table 4-39: ICU_CID Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	VALUE	Context ID Value. The ICU_CID.VALUE bits hold a software defined values specifying the software context ID.

System ID Register

The CEC_SID register contains the system ID of the interrupt, which has most recently been accepted by the processor core.When the core accepts the interrupt, the cores sends an interrupt acknowledge to the SCI, causing the SCI to update the value in its SEC_CSID register. For more information about interrupt system ID values, see the SCI section of the processor hardware reference.

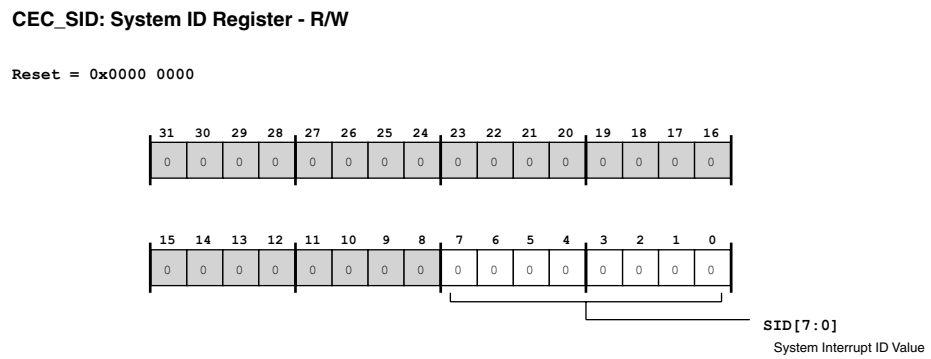


Figure 4-30: CEC_SID Register Diagram

Table 4-40: CEC_SID Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
7:0 (R/W)	SID	System Interrupt ID Value. The CEC_SID.SID bits hold the interrupt system ID value for the interrupt, which has been most recently accepted by the processor core.

ADSP-BF70x BP Register Descriptions

Branch Predictor (BP) contains the following registers.

Table 4-41: ADSP-BF70x BP Register List

Name	Description
BP_CFG	Configuration Register
BP_STAT	Status Register
BP_TAGIN	Tag Input Register
BP_TARGIN	Target Input Register
BP_TAG0	Tag 0 Register
BP_TAG1	Tag 1 Register
BP_TARG0	Target 0 Register
BP_TARG1	Target 1 Register

Configuration Register

The BP_CFG register configures branch predictor features, such as enabling dynamic branch prediction for various types of branch instructions, controlling updates to the branch prediction table, permitting access to entries in the prediction table and prediction table memory, and clearing the branch prediction table.

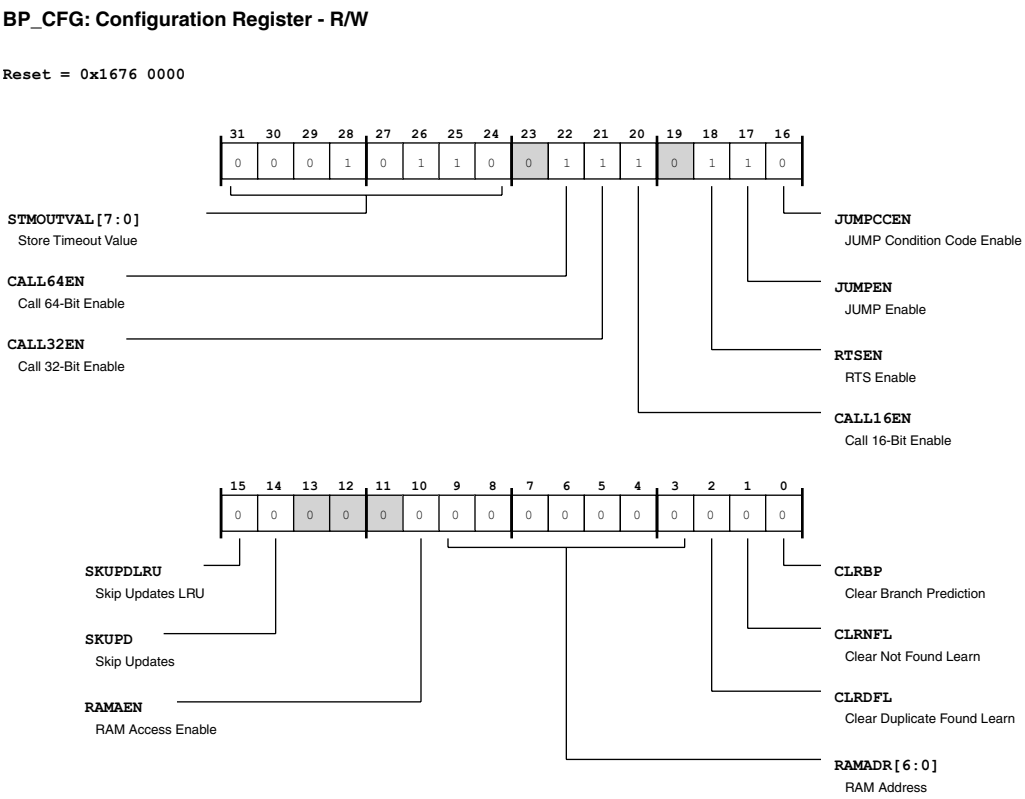


Figure 4-31: BP_CFG Register Diagram

Table 4-42: BP_CFG Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:24 (R/W)	STMOUTVAL	Store Timeout Value. The BP_CFG.STMOUTVAL bits select a timeout value (in CCLK cycles) for the wait status state for access requests to the store buffer.	
22 (R/W)	CALL64EN	Call 64-Bit Enable. The BP_CFG.CALL64EN bit enables branch prediction for CALL instructions encoded as 64-bit opcodes.	
		0	Disable (no prediction for CALL 64 bit)
		1	Enable (prediction for CALL 64 bit)
21 (R/W)	CALL32EN	Call 32-Bit Enable. The BP_CFG.CALL32EN bit enables branch prediction for CALL instructions encoded as 32-bit opcodes.	
		0	Disable (no prediction for CALL 32 bit)
		1	Enable (prediction for CALL 32 bit)

Table 4-42: BP_CFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
20 (R/W)	CALL16EN	Call 16-Bit Enable. The BP_CFG.CALL16EN bit enables branch prediction for CALL instructions encoded as 16-bit opcodes.	
		0	Disable (no prediction for CALL 16 bit)
		1	Enable (prediction for CALL 16 bit)
18 (R/W)	RTSEN	RTS Enable. The BP_CFG.RTSEN bit enables branch prediction for RTS instructions.	
		0	Disable (no prediction for RTS)
		1	Enable (prediction for RTS)
17 (R/W)	JUMPEN	JUMP Enable.	
		0	Disable (no prediction for unconditional JUMP)
		1	Enable (prediction for unconditional JUMP)
16 (R/W)	JUMPCCEN	JUMP Condition Code Enable. The BP_CFG.JUMPCCEN bit enable branch prediction for conditional JUMP instructions.	
		0	Disable (no prediction for conditional JUMP)
		1	Enable (prediction for conditional JUMP)
15 (R/W)	SKUPDLRU	Skip Updates LRU. The BP_CFG.SKUPDLRU bit directs the BP to skip updates to the branch prediction table if the prediction strength is Strongly Taken or is Not Taken and Primary is not LRU.	
		0	Disable (no update skip on ST or NT/LRU strength)
		1	Enable (skip updates on ST or NT/LRU strength)
14 (R/W)	SKUPD	Skip Updates. The BP_CFG.SKUPD bit directs the BP to skip updates to the branch prediction table if the prediction strength is Strongly Taken or is Not Taken.	
		0	Disable (no update skip on ST or NT strength)
		1	Enable (skip updates on ST or NT strength)
10 (R/W)	RAMAEN	RAM Access Enable. The BP_CFG.RAMAEN bit enables memory mapped register access to the branch prediction table in memory. While this access is enabled, dynamic branch prediction (uses the table) is disabled (operates as though the REGFILE_SYSCFG_BPEN bit =0).	
		0	Disable MMR accesses to branch prediction table
		1	Enable MMR accesses to branch prediction table

Table 4-42: BP_CFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
9:3 (R/W)	RAMADR	RAM Address. The BP_CFG.RAMADR bits provide the address value (within branch prediction memory) to access an entry in the branch prediction table. For read access, the addressed table entry is visible in the BP_TAG0/BP_TARG0 registers for way 0 and is visible in the BP_TAG1/BP_TARG1 registers for way 1. For write access, the write input uses the BP_TAGIN/BP_TARGIN registers. For example, a write to the BP_TAG0 register takes the value from BP_TAGIN and writes to the tag portion of prediction table way 0 location selected with the BP_CFG.RAMADR field.	
2 (R/W)	CLRDFL	Clear Duplicate Found Learn. The BP_CFG.CLRDFL bit clears the BP_STAT.DFL bit and keeps it cleared. When enabled, BP_CFG.CLRDFL prevents the branch predictor from reporting branch entry duplicate found errors.	
		0	Disable (normal DFL operation)
		1	Enable (clear DFL bit and keep it clear)
1 (R/W)	CLRNFL	Clear Not Found Learn. The BP_CFG.CLRNFL bit clears the BP_STAT.NFL bit and keeps it cleared. When enabled, BP_CFG.CLRNFL prevents the branch predictor from reporting branch entry not found errors.	
		0	Disable (normal NFL bit operation)
		1	Enable (clear NFL bit and keep clear)
0 (R0/W1A)	CLRBP	Clear Branch Prediction. The BP_CFG.CLRBP bit clears (W1A) the tag valid field for all the entries in the branch prediction table.	

Status Register

The BP_STAT register indicates the status for the branch predictor state machine, indicates the status of the store buffer, and indicates status for the current branch predictor operation.

BP_STAT: Status Register - R/W

Reset = 0x0000 0000

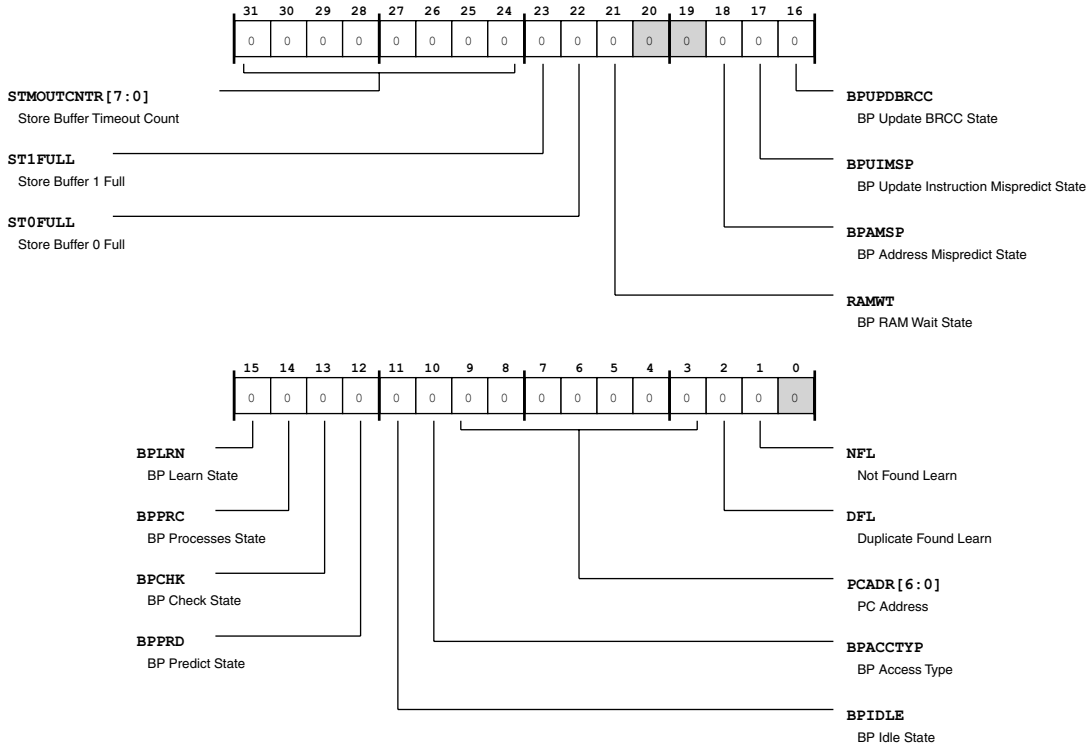


Figure 4-32: BP_STAT Register Diagram

Table 4-43: BP_STAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:24 (R/NW)	STMOUTCNTR	Store Buffer Timeout Count. The BP_STAT.STMOUTCNTR bits hold the value of the count remaining for the store buffer timeout. This field is automatically loaded when the BP_CFG.STMOUTVAL field is loaded and resets to that value when store buffer access is achieved before the count expires. The count decrements with each CCLK clock cycle while the branch predictor is in wait state (BP_STAT.RAMWT =1).	
23 (R/NW)	ST1FULL	Store Buffer 1 Full. The BP_STAT.ST1FULL bit indicates whether the branch predictor store buffer 1 is full.	
		0	No status
		1	Store buffer 1 full

Table 4-43: BP_STAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
22 (R/NW)	ST0FULL	Store Buffer 0 Full. The BP_STAT.ST0FULL bit indicates whether the branch predictor store buffer 0 is full.	
		0	No status
		1	Store buffer 0 full
21 (R/NW)	RAMWT	BP RAM Wait State. The BP_STAT.RAMWT bit indicates whether the branch predictor is in wait state (waiting for access to branch predictor memory).	
		0	No status
		1	Wait state
18 (R/NW)	BPAMSP	BP Address Mispredict State. The BP_STAT.BPAMSP bit indicates whether the branch predictor is in address mispredict state.	
		0	No status
		1	Address mispredict state
17 (R/NW)	BPUIMSP	BP Update Instruction Mispredict State. The BP_STAT.BPUIMSP bit indicates whether the branch predictor is in instruction mispredict state.	
		0	No status
		1	Instruction Mispredict state
16 (R/NW)	BPUPDBRCC	BP Update BRCC State. The BP_STAT.BPUPDBRCC bit indicates whether the branch predictor is in update BRCC state.	
		0	No status
		1	Updated BRCC state
15 (R/NW)	BPLRN	BP Learn State. The BP_STAT.BPLRN bit indicates whether the branch predictor is in learn state.	
		0	No status
		1	Learn state
14 (R/NW)	BPPRC	BP Processes State. The BP_STAT.BPPRC bit indicates whether the branch predictor is in process state.	
		0	No status
		1	Process state
13 (R/NW)	BPCHK	BP Check State. The BP_STAT.BPCHK bit indicates whether the branch predictor is in check state.	
		0	No status
		1	Check state

Table 4-43: BP_STAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
12 (R/NW)	BPPRD	BP Predict State. The BP_STAT.BPPRD bit indicates whether the branch predictor is in predict state.	
		0	No status
		1	Predict state
11 (R/NW)	BPIDLE	BP Idle State. The BP_STAT.BPIDLE bit indicates whether the branch predictor is in idle state.	
		0	No status
		1	Idle state
10 (R/NW)	BPACCTYP	BP Access Type. The BP_STAT.BPACCTYP bit indicates whether the most recent branch prediction table access was made for predicting or learning the branch.	
		0	Learning the branch
		1	Predicting the branch
9:3 (R/NW)	PCADR	PC Address. The BP_STAT.PCADR bits provide the address value (within branch prediction memory) of the entry in the branch prediction table corresponding to the most recent PC entry accessed.	
2 (R/NW)	DFL	Duplicate Found Learn. The BP_STAT.DFL bit indicates whether a duplicate branch prediction table entry was found during a BP learn state.	
		0	No status
		1	Entry duplicate found
1 (R/NW)	NFL	Not Found Learn. The BP_STAT.NFL bit indicates whether a branch prediction table entry was not found during a BP learn state.	
		0	No status
		1	Entry not found

Tag Input Register

The BP_TAGIN register holds data for write entry operations to the branch prediction table. The value in BP_TAGIN is written to the tag portion of prediction table at the location defined by the BP_CFG.RAMADR field for way 0 (if a write occurs to BP_TAG0) or way 1 (if a write occurs to BP_TAG1).

BP_TAGIN: Tag Input Register - R/W

Reset = 0x0000 0000

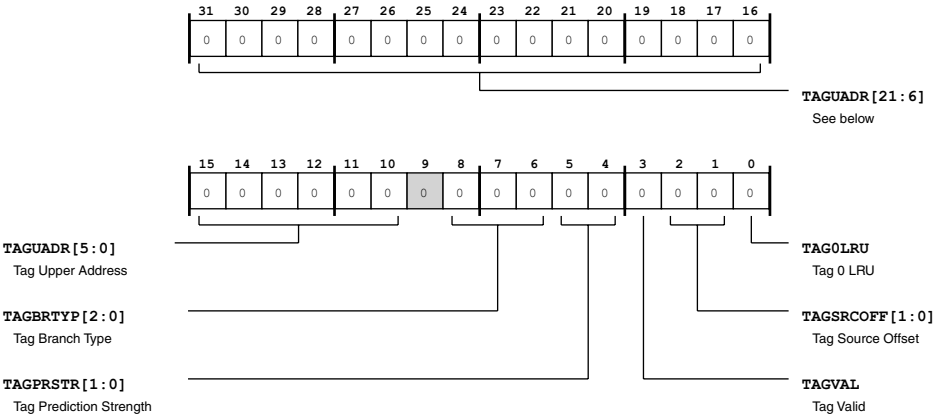


Figure 4-33: BP_TAGIN Register Diagram

Table 4-44: BP_TAGIN Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:10 (R/W)	TAGUADR	Tag Upper Address. The BP_TAGIN.TAGUADR bits hold bits 31-10 of the source address for the branch predicted by this tag.	
8:6 (R/W)	TAGBRTYP	Tag Branch Type. The BP_TAGIN.TAGBRTYP bits indicate the branch type for the branch predicted by this tag.	
		0	JUMP conditional
		1	JUMP unconditional
		2	RTS (return from subroutine)
		4	CALL 16-bit opcode
		5	CALL 32-bit opcode
		6	CALL 64-bit opcode
5:4 (R/W)	TAGPRSTR	Tag Prediction Strength. The BP_TAGIN.TAGPRSTR bits indicate the prediction strength for the branch predicted by this tag.	
		0	Strongly Not Taken
		1	Weakly Not Taken
		2	Weakly Taken
		3	Strongly Taken
3 (R/W)	TAGVAL	Tag Valid. The BP_TAGIN.TAGVAL bit indicates (if set, =1) the prediction associated with this tag is valid.	

Table 4-44: BP_TAGIN Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
2:1 (R/W)	TAGSRCOFF	Tag Source Offset. The BP_TAGIN.TAGSRCOFF bits hold bits 2-1 of the source address for the branch predicted by this tag.
0 (R/W)	TAG0LRU	Tag 0 LRU. The BP_TAGIN.TAG0LRU bit provides the least recently used (LRU) flag for TAG 0.

Target Input Register

The BP_TARGIN register holds data for write entry operations to the branch prediction table. The value in BP_TARGIN is written to the target portion of prediction table at the location defined by the BP_CFG.RAMADR field for way 0 (if a write occurs to BP_TARG0) or way 1 (if a write occurs to BP_TARG1).

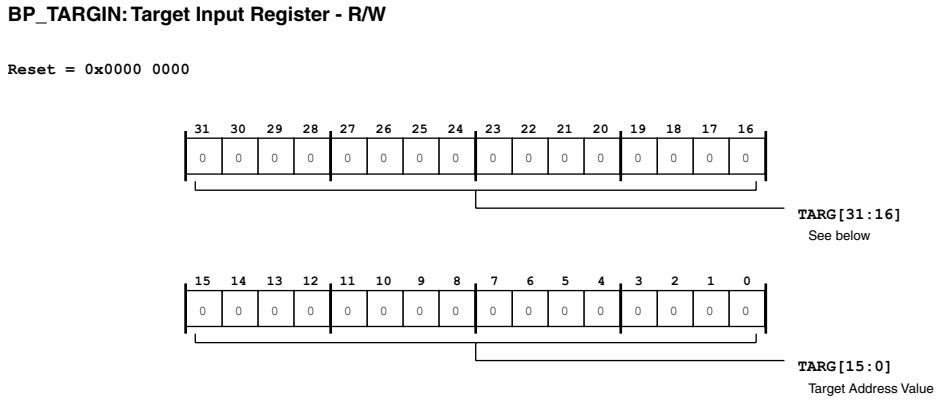


Figure 4-34: BP_TARGIN Register Diagram

Table 4-45: BP_TARGIN Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	TARG	Target Address Value. The BP_TARGIN.TARG bits hold a branch target address value for input (write) to way 0 or 1.

Tag 0 Register

The BP_TAG0 register provides read/write access to way 0 of the branch prediction table.

For write operations, the value in BP_TAGIN is written to the tag portion of prediction table at the location defined by the BP_CFG.RAMADR field for way 0 when a write occurs to BP_TAG0.

For read operations, a read of the BP_TAG0 register returns the tag portion from the previous read of the prediction table at the location defined by the BP_CFG.RAMADR field for way 0.

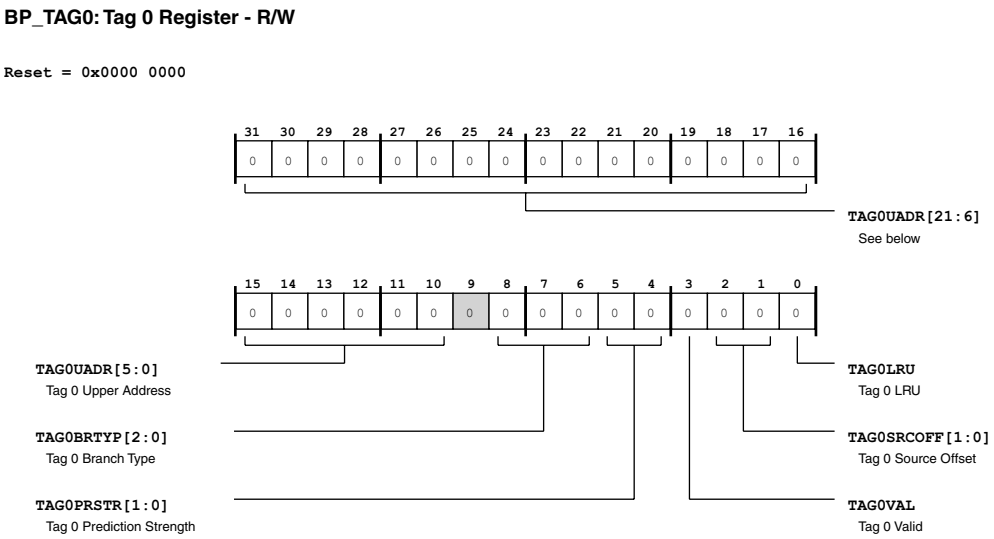


Figure 4-35: BP_TAG0 Register Diagram

Table 4-46: BP_TAG0 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:10 (R/NW)	TAG0UADR	Tag 0 Upper Address. The BP_TAG0.TAG0UADR bits hold bits 31-10 of the source address for the branch predicted by this tag.	
8:6 (R/NW)	TAG0BRTYP	Tag 0 Branch Type. The BP_TAG0.TAG0BRTYP bits indicate the branch type for the branch predicted by this tag.	
		0	JUMP conditional
		1	JUMP unconditional
		2	RTS (return from subroutine)
		4	CALL 16-bit opcode
		5	CALL 32-bit opcode
		6	CALL 64-bit opcode
5:4 (R/NW)	TAG0PRSTR	Tag 0 Prediction Strength. The BP_TAG0.TAG0PRSTR bits indicate the prediction strength for the branch predicted by this tag.	
		0	Strongly Not Taken
		1	Weakly Not Taken
		2	Weakly Taken
		3	Strongly Taken

Table 4-46: BP_TAG0 Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
3 (R/NW)	TAG0VAL	Tag 0 Valid. The BP_TAG0 bit indicates (if set, =1) the prediction associated with this tag is valid.
2:1 (R/NW)	TAG0SRCOFF	Tag 0 Source Offset. The BP_TAG0.TAG0SRCOFF bits hold bits 2-1 of the source address for the branch predicted by this tag.
0 (R/NW)	TAG0LRU	Tag 0 LRU. The BP_TAG0.TAG0LRU bit provides the least recently used (LRU) flag for TAG 0.

Tag 1 Register

The BP_TAG1 register provides read/write access to way 1 of the branch prediction table.

For write operations, the value in BP_TAGIN is written to the tag portion of prediction table at the location defined by the BP_CFG.RAMADR field for way 1 when a write occurs to BP_TAG1.

For read operations, a read of the BP_TAG1 register returns the tag portion from the previous read of the prediction table at the location defined by the BP_CFG.RAMADR field for way 1.

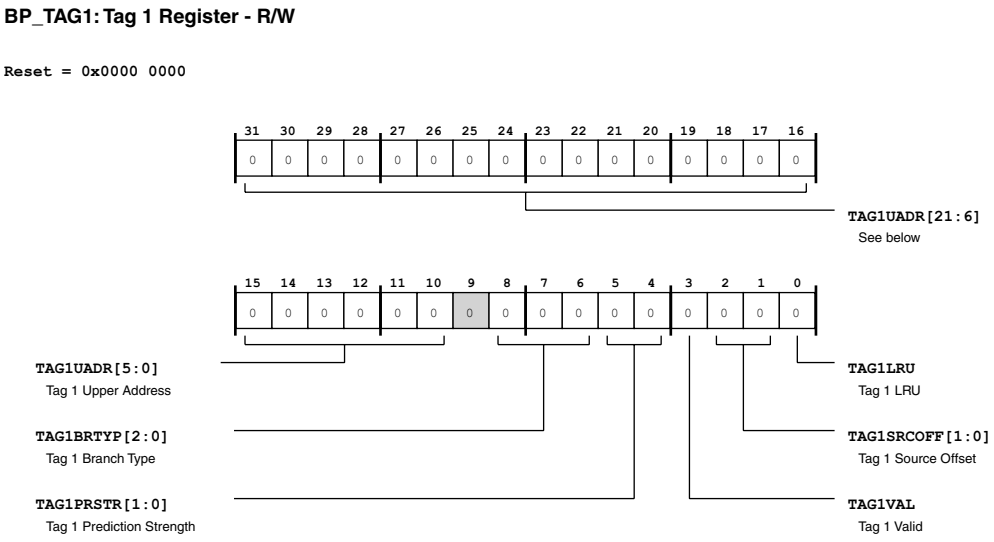


Figure 4-36: BP_TAG1 Register Diagram

Table 4-47: BP_TAG1 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:10 (R/NW)	TAG1UADR	Tag 1 Upper Address. The BP_TAG1.TAG1UADR bits hold bits 31-10 of the source address for the branch predicted by this tag.

Table 4-47: BP_TAG1 Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
8:6 (R/NW)	TAG1BRTYP	Tag 1 Branch Type. The BP_TAG1.TAG1BRTYP bits indicate the branch type for the branch predicted by this tag.	
		0	JUMP conditional
		1	JUMP unconditional
		2	RTS (return from subroutine)
		4	CALL 16-bit opcode
		5	CALL 32-bit opcode
		6	CALL 64-bit opcode
5:4 (R/NW)	TAG1PRSTR	Tag 1 Prediction Strength. The BP_TAG1.TAG1PRSTR bits indicate the prediction strength for the branch predicted by this tag.	
		0	Strongly Not Taken
		1	Weakly Not Taken
		2	Weakly Taken
		3	Strongly Taken
3 (R/NW)	TAG1VAL	Tag 1 Valid. The BP_TAG1.TAG1VAL bit indicates (if set, =1) the prediction associated with this tag is valid.	
2:1 (R/NW)	TAG1SRCOFF	Tag 1 Source Offset. The BP_TAG1.TAG1SRCOFF bits hold bits 2-1 of the source address for the branch predicted by this tag.	
0 (R/NW)	TAG1LRU	Tag 1 LRU. The BP_TAG1.TAG1LRU bit provides the least recently used (LRU) flag for TAG 1.	

Target 0 Register

The BP_TARG0 register provides read/write access to way 0 of the branch prediction table.

For write operations, the value in BP_TARGIN is written to the target portion of prediction table at the location defined by the BP_CFG.RAMADR field for way 0 when a write occurs to BP_TARG0.

For read operations, a read of the BP_TARG0 register returns the target portion from the previous read of the prediction table at the location defined by the BP_CFG.RAMADR field for way 0.

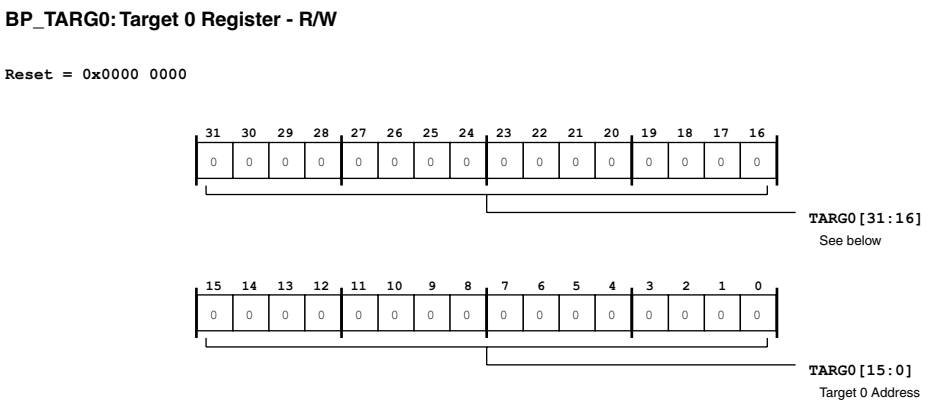


Figure 4-37: BP_TARG0 Register Diagram

Table 4-48: BP_TARG0 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/NW)	TARG0	Target 0 Address. The BP_TARG0.TARG0 bits hold a branch target 0 address value for way 0.

Target 1 Register

The BP_TARG1 register provides read/write access to way 1 of the branch prediction table.

For write operations, the value in BP_TARGIN is written to the target portion of prediction table at the location defined by the BP_CFG.RAMADR field for way 1 when a write occurs to BP_TARG1.

For read operations, a read of the BP_TARG1 register returns the target portion from the previous read of the prediction table at the location defined by the BP_CFG.RAMADR field for way 1.

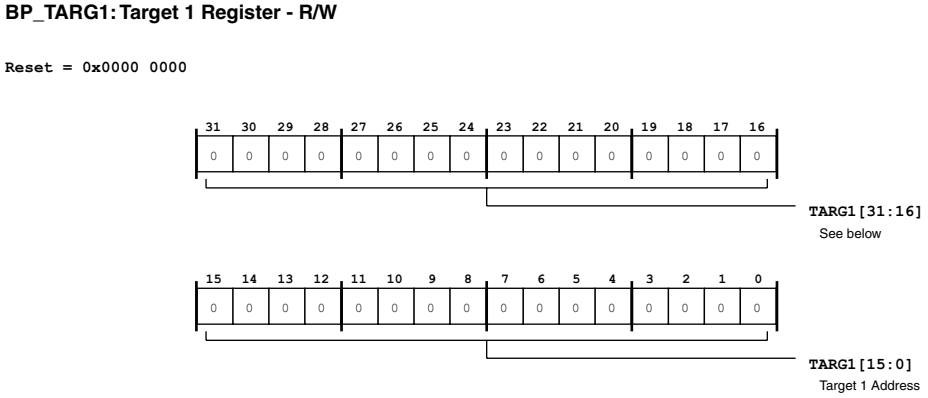


Figure 4-38: BP_TARG1 Register Diagram

Table 4-49: BP_TARG1 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/NW)	TARG1	Target 1 Address. The BP_TARG1.TARG1 bits hold a branch target 1 address value for way 1.

5 Core Timer (TMR)

Each processor core has its own dedicated timer. The core timer is clocked by the internal processor clock and is typically used as a system tick clock for generating periodic operating system interrupts.

NOTE: The core timer stops counting when there is an emulation event. The emulation event is controlled by the SDU (System Debug Unit). For more information, see the SDU chapter.

TMR Features

The core timer is a programmable 32-bit interval timer which can generate periodic interrupts. Unlike other peripherals, the core timer resides inside the Blackfin core and runs at the core clock (CCLK) rate. Core timer features include:

- 32-bit timer with 8-bit prescaler
- Operates at core clock (*CCLK*) rate
- Dedicated high-priority interrupt channel
- Single-shot or continuous operation

TMR Functional Description

The TMR (core timer) is a programmable 32-bit interval timer in each processor core. The following sections describe the TMR features:

- ADSP-BF70x TMR Register List
- TMR Block Diagram

ADSP-BF70x TMR Register List

Table 5-1: ADSP-BF70x TMR Register List

Name	Description
TCNTL	Timer Control Register

Table 5-1: ADSP-BF70x TMR Register List (Continued)

Name	Description
TPERIOD	Timer Period Register
TSCALE	Timer Scale Register
TCOUNT	Timer Count Register

TMR Block Diagram

The following figure shows the core timer block diagram.

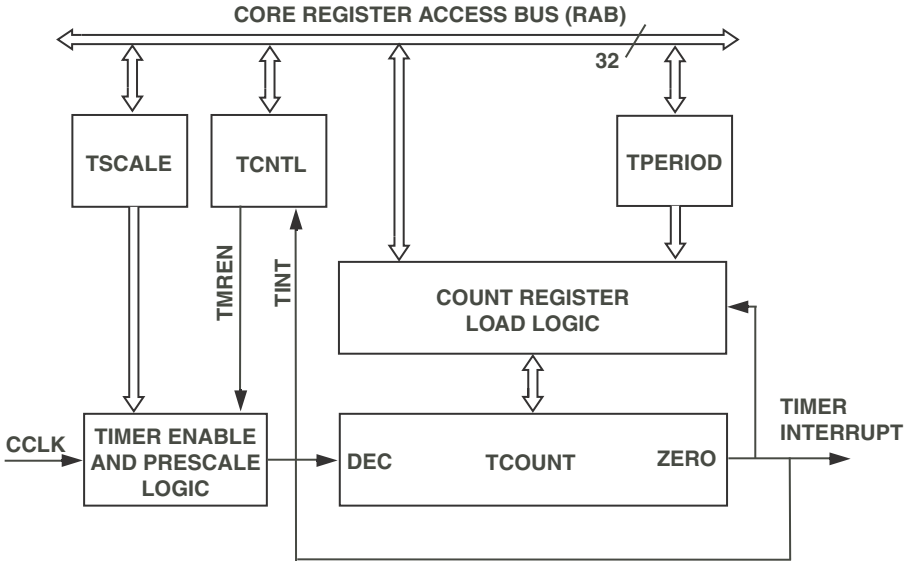


Figure 5-1: Core Timer Block Diagram

External Interfaces

The core timer does not directly interact with any pins of the chip.

Internal Interfaces

The core timer is accessed through the 32-bit register access bus. The module is clocked by the core clock *CCLK*. The timer's dedicated interrupt request is a higher priority than requests from all other peripherals.

TMR Operation

The software should initialize the TCOUNT register before the timer is enabled. The TCOUNT register can be written directly, but writes to the TPERIOD register are also passed through to TCOUNT.

When the timer is enabled by setting the TCNTL.EN bit, the TCOUNT register is decremented once every time the prescaler register (TSCALE) expires, that is, every TSCALE + 1 number of *CCLK* clock cycles. When the value of the TCOUNT register reaches 0, an interrupt is generated and the TCNTL.INT bit is set.

If the TCNTL.AUTORLD bit is set, then the TCOUNT register is reloaded with the contents of the TPERIOD register and the count begins again. If the TCNTL.AUTORLD bit is not set, the timer stops operation.

The core timer can be put into low power mode by clearing the TCNTL.PWR bit. Before using the timer, set the TCNTL.PWR bit. This restores clocks to the timer unit. When TCNTL.PWR is set, the core timer may then be enabled by setting the TCNTL.EN bit.

NOTE: Hardware behavior is undefined if TCNTL.EN is set when TCNTL.PWR=0.

Interrupt Processing

The timer's dedicated interrupt request is a higher priority than requests from all other peripherals. The request goes directly to the core event controller (CEC) and does not pass through the system event controller (SEC). Therefore, the interrupt processing is also completely in the *CCLK* domain.

NOTE: The core timer interrupt request is edge-sensitive and cleared by hardware automatically as soon as the interrupt is serviced.

The TCNTL.INT bit indicates that an interrupt has been generated and programs need to write a 0 (not W1C) to clear it (the write is optional). The core time module doesn't provide any further interrupt enable bit. When the timer is enabled, interrupts can be masked in the CEC controller.

ADSP-BF70x TMR Register Descriptions

Timer (TMR) contains the following registers.

Table 5-2: ADSP-BF70x TMR Register List

Name	Description
TCNTL	Timer Control Register
TPERIOD	Timer Period Register
TSCALE	Timer Scale Register
TCOUNT	Timer Count Register

Timer Control Register

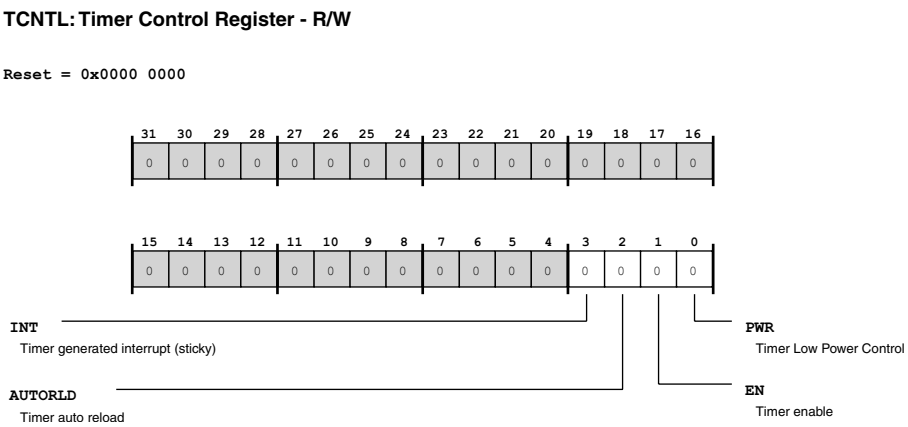


Figure 5-2: TCNTL Register Diagram

Table 5-3: TCNTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
3 (R/W)	INT	Timer generated interrupt (sticky).	
		0	No interrupt has been generated
		1	Interrupt has been generated
2 (R/W)	AUTORLD	Timer auto reload.	
1 (R/W)	EN	Timer enable.	
		0	Disable
		1	Enable
0 (R/W)	PWR	Timer Low Power Control.	
		0	Low power mode
		1	Active state

Timer Period Register

TPERIOD: Timer Period Register - R/W

Reset = 0x0000 0000

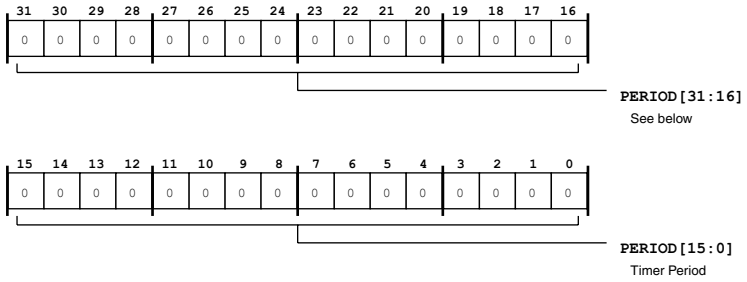


Figure 5-3: TPERIOD Register Diagram

Table 5-4: TPERIOD Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	PERIOD	Timer Period.

Timer Scale Register

TSCALE: Timer Scale Register - R/W

Reset = 0x0000 0000

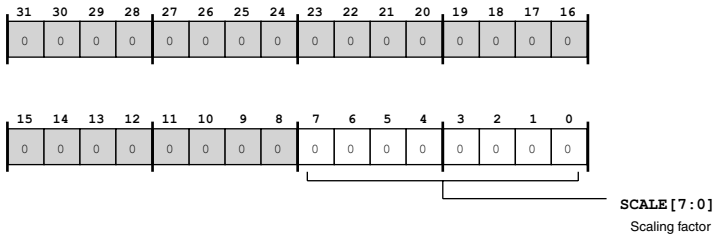


Figure 5-4: TSCALE Register Diagram

Table 5-5: TSCALE Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
7:0 (R/W)	SCALE	Scaling factor.

Timer Count Register

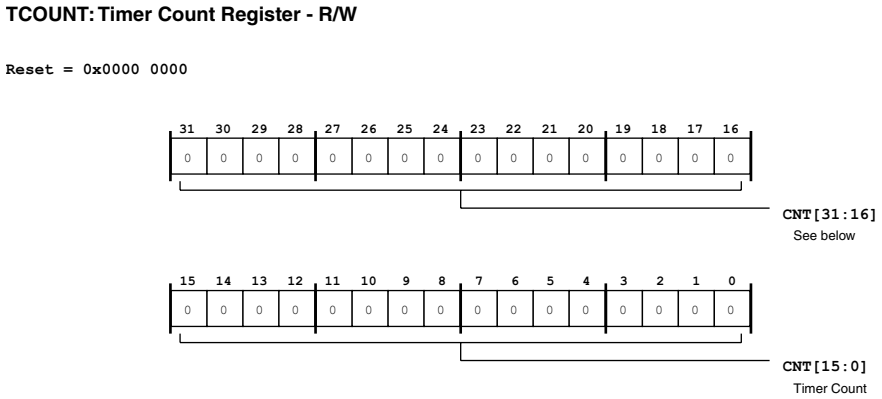


Figure 5-5: TCOUNT Register Diagram

Table 5-6: TCOUNT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	Timer Count.

6 Address Arithmetic Unit

Like most DSP (digital signal processor) and RISC (reduced instruction set computer) platforms, the Blackfin processors have a load/store architecture. Computation operands and results are always represented by core registers. Prior to computation, data is loaded from memory into core registers and results are stored back by explicit move operations. The Address Arithmetic Unit (AAU) provides all the required support to keep data transport between memory and core registers efficient and seamless. Having a separate arithmetic unit for address calculations prevents the data computation block from being burdened by address operations. Not only can the store operations, load and store operations occur in parallel operations parallel to data computations, but memory addresses can also be calculated at the same time.

The AAU uses Data Address Generators (DAGs) to generate addresses for data moves to and from memory. By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address. The figure shows the AAU block diagram.

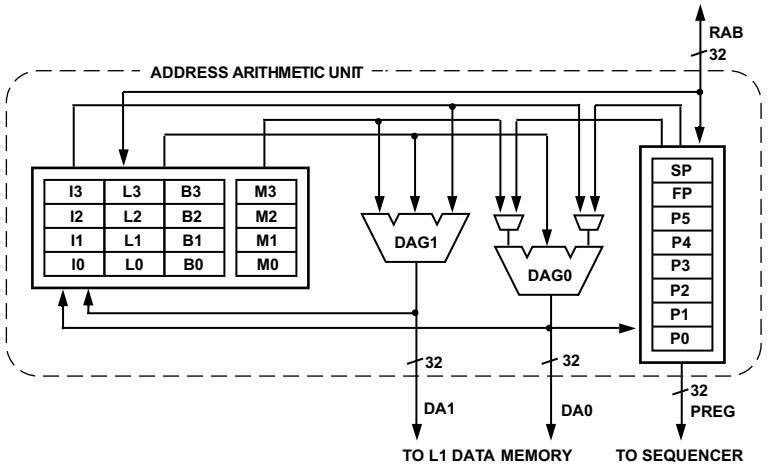


Figure 6-1: AAU Block Diagram

The AAU architecture supports several functions that minimize overhead in data access routines. These functions include:

- Supply address - Provides an address during a data access
- Supply address and post-modify - Provides an address during a data move and auto-increments/decrements the stored address for the next move
- Supply address with offset - Provides an address from a base with an offset without incrementing the original address pointer
- Modify address - Increments or decrements the stored address without performing a data move

- Bit-reversed carry address - Provides a bit-reversed carry address during a data move without reversing the stored address

The AAU comprises two DAGs, nine Pointer registers, four Index registers and four complete sets of related Modify, Base, and Length registers. These registers (shown in the AAU figure) hold the values that the DAGs use to generate addresses. The types of registers are:

- Index registers, `I[3:0]`. Unsigned 32-bit Index registers hold an address pointer to memory. For example, the instruction `R3 = [I0]` loads the data value found at the memory location pointed to by the register `I0`. Index registers can be used for 16- and 32-bit memory accesses.
- Modify registers, `M[3:0]`. Signed 32-bit Modify registers provide the increment or step size by which an Index register is post-modified during a register move. For example, the `R0 = [I0 ++ M1]` instruction directs the DAG to: - Output the address in register `I0` - Load the contents of the memory location pointed to by `I0` into `R0` - Modify the contents of `I0` by the value contained in the `M1` register
- Base and Length registers, `B[3:0]` and `L[3:0]`. Unsigned 32-bit Base and Length registers set up the range of addresses and the starting address of a buffer. Each `B`, `L` pair is always coupled with a corresponding index register, for example, `I3`, `B3`, `L3`. For more information on circular buffers, see Addressing Circular Buffers.
- Pointer registers, `P[5:0]`, `FP`, `USP`, and `SP`. 32-bit Pointer registers hold an address pointer to memory. The `P[5:0]` field, `FP` (Frame Pointer) and `SP/USP` (Stack Pointer/User Stack Pointer) can be manipulated and used in various instructions. For example, the instruction `R3 = [P0]` loads the register `R3` with the data value found at the memory location pointed to by the register `P0`. The Pointer registers have no effect on circular buffer addressing. They can be used for 8-, 16-, and 32-bit memory accesses. For added mode protection, `SP` is accessible only in Supervisor mode, while `USP` is accessible in User mode.

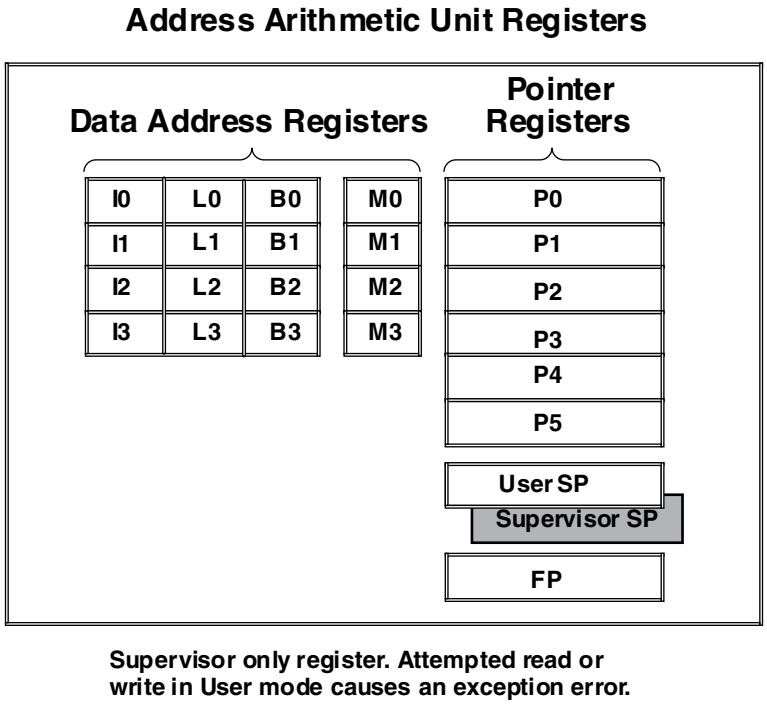


Figure 6-2: Address Arithmetic Unit

Addressing With the AAU

The DAGs can generate an address that is incremented by a value or by a register. In post-modify addressing, the DAG outputs the index register value unchanged; then the DAG adds an modify register or immediate value to the index register.

In indexed addressing, the DAG adds a small offset to the value in the pointer register, but does not update the pointer register with this new value, thus providing an offset for that particular memory access.

In direct addressing the entire address is specified in the instruction, and does not depend upon the value in any register.

The processor is byte addressed. Depending on the type of data used, increments and decrements to the address registers can be by 1, 2, or 4 to match the 8-, 16-, or 32-bit accesses.

For example, consider the following instruction:

```
R0 = [ P3++ ];
```

This instruction fetches a 32-bit word, pointed to by the value in P3, and places it in R0. It then post-increments P3 by four to point to the data after the word that has been fetched.

```
R0.L = W [ I3++ ];
```

This instruction fetches a 16-bit word, pointed to by the value in I3, and places it in the low half of the destination register, R0.L. It then post-increments I3 by two.

```
R0 = B [ P3++ ] (Z) ;
```

This instruction fetches an 8-bit word, pointed to by the value in P3, and places it in the destination register, R0. It then post-increments P3 by one. The byte value may be zero extended (as shown) or sign extended into the 32-bit data register.

Instructions using Index registers use an modify register or a small immediate value (+/- 2 or 4) as the modifier. Instructions using Pointer registers use a small immediate value or another pointer register as the modifier. For details, see AAU Instruction Summary.

There are no restrictions on data alignment. A 32-bit word can be fetched from any address and the four bytes from the specified address and the next three numerically increasing addresses are fetched. Similarly a 16-bit load may be from ant two adjacent addresses. The byte order is little-endian so the lower addressed bytes always contain the lest significant bits of the value.

Pointer Register File

The general-purpose Address Pointer registers, also called pointer registers (or Preg), are organized as:

- 6-entry, pointer register file P[5:0](see Pointer Register)
- Frame Pointer (FP) used to point to the current procedure's activation record (see Frame Pointer Register)
- Stack Pointer (SP) used to point to the last used location on the runtime stack (see Stack Pointer Register)

Pointer registers are 32 bits wide. Although pointer registers are primarily used for address calculations, they may also be used for general integer arithmetic with a limited set of arithmetic operations; for instance, to maintain counters. However, unlike the Data registers, pointer register arithmetic does not affect the Arithmetic Status (ASTAT) register status bits.

Frame and Stack Pointers

In many respects, the Frame and Stack Pointer registers perform like the other pointer registers, P[5:0]. They can act as general pointers in any of the load/store instructions, for example, R1 = B[SP] (Z). However, FP and SP have additional functionality. For more information, see Frame Pointer Register,

Stack Pointer Register, and User Stack Pointer Register.

The Stack Pointer registers include:

- a User Stack Pointer (`USP` in Supervisor mode, `SP` in User mode)
- a Supervisor Stack Pointer (`SP` in Supervisor mode)

The User Stack Pointer register and the Supervisor Stack Pointer register are accessed using the register alias `SP`. Depending on the current processor operating mode, only one of these registers is active and accessible as `SP`:

- In User mode, any reference to `SP` (for example, stack pop `R0 = [ SP++ ] ;`) implicitly uses the `USP` as the effective address.
- In Supervisor mode, the same reference to `SP` (for example, `R0 = [ SP++ ] ;`) implicitly uses the Supervisor Stack Pointer as the effective address. To manipulate the User Stack Pointer for code running in Supervisor mode, use the register alias `USP`. When in Supervisor mode, a register move from `USP` (for example, `R0 = USP ;`) moves the current User Stack Pointer into `R0`. The register alias `USP` can only be used in Supervisor mode.

Some load/store instructions use `FP` and `SP` implicitly:

- `FP`-indexed load/store, which extends the addressing range for 16-bit encoded load/stores
- Stack push/pop instructions, including those for pushing and popping multiple registers
- Link/unlink instructions, which control stack frame space and manage the Frame Pointer register (`FP`) for that space

DAG Register Set

DSP instructions primarily use the Data Address Generator (DAG) register set for addressing. The data address register set consists of these registers:

- `I[3:0]` contain index addresses (see Index (Circular Buffer) Register)
- `M[3:0]` contain modify values (see Modify (Circular Buffer) Register)
- `B[3:0]` contain base addresses (see Base (Circular Buffer) Register)
- `L[3:0]` contain length values (see Length (Circular Buffer) Register)

All data address registers are 32 bits wide.

The I (Index) registers and B (Base) registers always contain addresses of 8-bit bytes in memory. The Index registers contain an effective address. The M (Modify) registers contain an offset value that is added to one of the Index registers or subtracted from it.

The B and L (Length) registers define circular buffers. The B register contains the starting address of a buffer, and the L register contains the length in bytes. Each L and B register pair is associated with the

corresponding I register. For example, L0 and B0 are always associated with I0. However, any M register may be associated with any I register. For example, I0 may be modified by M3.

Indexed Addressing With Index and Pointer Registers

Indexed addressing uses the value in the Index or Pointer register as an effective address. This instruction can load or store 16- or 32-bit values. The default is a 32-bit transfer. If a 16-bit transfer is required, then the W (16-bit word) designator is used to preface the load or store.

For example:

```
R0 = [ I2 ];
```

loads a 32-bit value from an address pointed to by I2 and stores it in the destination register R0.

```
R0.H = W [ I2 ];
```

loads a 16-bit value from an address pointed to by I2 and stores it in the 16-bit destination register R0.H.

```
[ P1 ] = R0 ;
```

is an example of a 32-bit store operation.

Pointer registers can be used for 8-bit loads and stores.

For example:

```
B [ P1 ] = R0 ;
```

stores the 8-bit value from the R0 register in the address pointed to by the P1 register.

Loads With Zero or Sign Extension

When a 32-bit register is loaded by an 8-bit or 16-bit memory read, the value can be extended to the full register width. A trailing Z character in parenthesis is used to zero-extend the loaded value. An X character forces sign extension. The following examples assume that P1 points to a memory location that contains a value of 0x8080.

```
R0 = W[P1] (Z) ;   /*  R0 = 0x0000 8080 */

R1 = W[P1] (X) ;   /*  R1 = 0xFFFF 8080 */

R2 = B[P1] (Z) ;   /*  R2 = 0x0000 0080 */

R3 = B[P1] (X) ;   /*  R3 = 0xFFFF FF80 */
```

Indexed Addressing With Immediate Offset

Indexed addressing allows programs to obtain values from data tables, with reference to the base of that table. The Pointer register is modified by the immediate field and then used as the effective address. The value of the Pointer register is not updated.

For example, if `P1 = 0x13`, then `[P1 + 0x11]` would effectively be equal to `[0x24]`.

Auto-increment and Auto-decrement Addressing

Auto-increment addressing updates the Pointer and Index registers after the access. The amount of increment depends on the word size. An access of 32-bit words results in an update of the Pointer by 4. A 16-bit word access updates the Pointer by 2, and an access of an 8-bit word updates the Pointer by 1. Both 8- and 16-bit read operations may specify either to sign-extend or zero-extend the contents into the destination register. Pointer registers may be used for 8-, 16-, and 32-bit accesses while Index registers may be used only for 16- and 32-bit accesses.

For example:

```
R0 = W [ P1++ ](Z);
```

loads a 16-bit word into a 32-bit destination register from an address pointed to by the `P1` Pointer register. The Pointer is then incremented by 2 and the word is zero extended to fill the 32-bit destination register.

Auto-decrement works the same way by decrementing the address after the access.

For example:

```
R0 = [ I2-- ] ;
```

loads a 32-bit value into the destination register and decrements the Index register by 4.

Pre-modify Stack Pointer Addressing

The only pre-modify instruction in the processor uses the Stack Pointer register, `SP`. The address in `SP` is decremented by 4 and then used as an effective address for the store. The instruction `[ --SP ] = R0;` is used for stack push operations and can support only a 32-bit word transfer.

Post-modify Addressing

Post-modify addressing uses the value in the Index or Pointer registers as the effective address and then modifies it by the contents of another register. Pointer registers are modified by other Pointer registers. Index registers are modified by Modify registers. Post-modify addressing does not support the Pointer registers as destination registers, nor does it support byte-addressing.

For example:

```
R5 = [ P1++P2 ] ;
```

loads a 32-bit value into the R5 register, found in the memory location pointed to by the P1 register.

The value in the P2 register is then added to the value in the P1 register.

For example:

```
R2 = W [ P4++P5 ] (Z) ;
```

loads a 16-bit word into the low half of the destination register R2 and zero-extends it to 32 bits. The value of the pointer P4 is incremented by the value of the pointer P5.

For example:

```
R2 = [ I2++M1 ];
```

loads a 32-bit word into the destination register R2. The value in the Index register, I2, is updated by the value in the Modify register, M1.

Direct Addressing

Direct addressing uses the immediate field from the instruction as the effective address. The location addressed does not depend upon the contents of any register. The source or destination may be a Pointer register, a Data register or a Data register half. Both 8- and 16-bit read operations may specify either to sign-extend or zero-extend the value into the destination register.

For example:

```
[ 0x100 ] = SP ;
```

Stores the stack pointer register in the 32-bit word at address 0x100.

Typically the address is specified as a symbolic value defined in the assembler.

```
R0 = B [ myvar ] (Z) ;
```

The preceding instruction loads a byte from an address identified by the symbolic value myvar which might be defined with a .VAR directive in the assembly file.

Direct address instructions are convenient for accessing memory mapped registers which are always at a well known address. They also enable all registers to be saved without modifying any of them.

For example a CPLB miss handler could be written to switch to a private stack in a known safe region of memory to avoid the case where the stack pointer itself had caused the CPLB miss.

```
.SECTION scratchpad;
.ALIGN 4;
.VAR save_sp, safe_stack[BIG_ENOUGH];
...
[ save_sp ] = SP;    // save stack pointer
```

```
SP = safe_stack+BIG_ENOUGH; // make it point to the safe stack
// now registers can be saved into the safe area
[ --SP ] = ( R7:5, P5:P4 );
[ --SP ] = ASTAT;
...
// restore registers
ASTAT = [SP++];
( R7:5, P5:P4 ) = [ SP++ ];
SP = [ save_sp ];    // restore stack pointer
RTE;
```

Addressing Circular Buffers

The DAGs support addressing circular buffers. Circular buffers are a range of addresses containing data that the DAG steps through repeatedly, wrapping around to repeat stepping through the same range of addresses in a circular pattern.

The DAGs use four types of data address registers for addressing circular buffers. For circular buffering, the registers operate this way:

- The Index (I) register contains the value that the DAG outputs on the address bus. For more information, see Index (Circular Buffer) Register.
- The Modify (M) register contains the post-modify amount (positive or negative) that the DAG adds to the index register at the end of each memory access. Any modify register can be used with any index register. The modify value can also be an immediate value instead of an modify register. The size of the modify value must be less than or equal to the length (length register) of the circular buffer. For more information, see Modify (Circular Buffer) Register.
- The Length (L) register sets the size of the circular buffer and the address range through which the DAG circulates the index register. L is positive and cannot have a value greater than 2^{32} - 1. If an length register's value is zero, its circular buffer operation is disabled. For more information, see Length (Circular Buffer) Register.
- The Base (B) register or the base register plus the length register is the value with which the DAG compares the modified index register value after each access. For more information, see Base (Circular Buffer) Register.

To address a circular buffer, the DAG steps the Index pointer (index register) through the buffer values, post-modifying and updating the index on each access with a positive or negative modify value from the modify register.

If the Index pointer falls outside the buffer range, the DAG subtracts the length of the buffer (length register) from the value or adds the length of the buffer to the value, wrapping the Index pointer back to a point inside the buffer.

The starting address that the DAG wraps around is called the buffer's base address (base register). There are no restrictions on the value of the base address for circular buffers that contains 8-bit data. Circular buffers that contain 16- and 32-bit data must be 16-bit aligned and 32-bit aligned, respectively. Exceptions

can be made for video operations. For more information, see Memory Address Alignment. Circular buffering uses post-modify addressing.

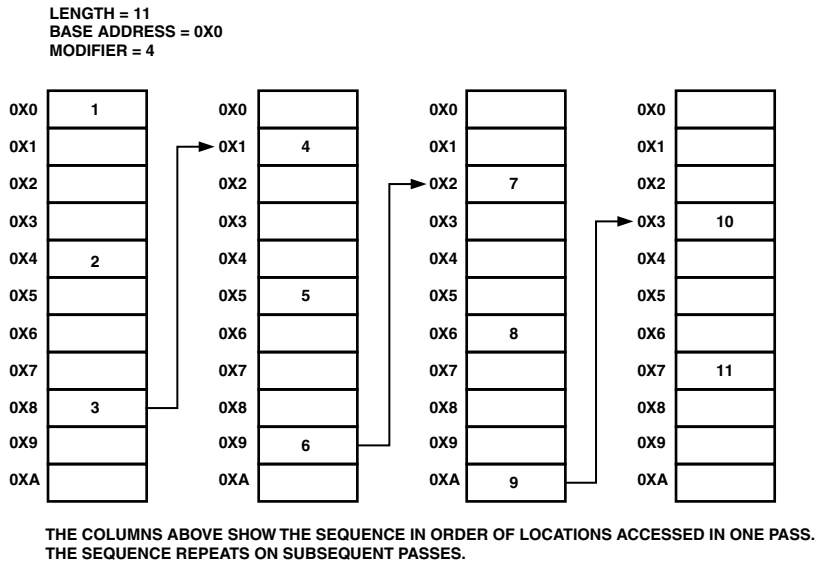


Figure 6-3: Circular Data Buffers

As seen in the **Circular Data Buffers** figure, on the first post-modify access to the buffer, the DAG outputs the index register value on the address bus, then modifies the address by adding the modify value.

- If the updated index value is within the buffer length, the DAG writes the value to the index register.
- If the updated index value exceeds the buffer length, the DAG subtracts (for a positive modify value) or adds (for a negative modify value) the length register value before writing the updated index value to the index register.

In equation form, these post-modify and wraparound operations work as follows, shown for "I+M" operations.

- If M is positive: $I_{new} = I_{old} + M$ if $I_{old} + M <$ buffer base + length (end of buffer) $I_{new} = I_{old} + M - L$ if $I_{old} + M$ buffer base + length (end of buffer)
- If M is negative: $I_{new} = I_{old} + M$ if $I_{old} + M$ buffer base (start of buffer) $I_{new} = I_{old} + M + L$ if $I_{old} + M <$ buffer base (start of buffer)

Addressing With Bit-reversed Addresses

To obtain results in sequential order, programs need bit-reversed carry addressing for some algorithms, particularly Fast Fourier Transform (FFT) calculations. To satisfy the requirements of these algorithms, the DAG's bit-reversed addressing feature permits repeatedly subdividing data sequences and storing this data in bit-reversed order. For detailed information about bit-reversed addressing, see the description of the modify/increment instruction.

Modifying Index and Pointer Registers

The DAGs support operations that modify an address value in an Index register without outputting an address. The operation, address-modify, is useful for maintaining pointers.

The address-modify operation modifies addresses in any Index and Pointer register (`I[3:0]`, `P[5:0]`, `FP`, `SP`) without accessing memory. If the index register's corresponding base and length registers are set up for circular buffering, the address-modify operation performs the specified buffer wraparound (if needed).

The syntax is similar to post-modify addressing (*`index += modifier`*). For Index registers, an modify register is used as the modifier. For Pointer registers, another pointer register is used as the modifier.

Consider the example, `I1 += M2` ;

This instruction adds `M2` to `I1` and updates `I1` with the new value.

Memory Address Alignment

The processor requires proper memory alignment to be maintained for the data size being accessed. Unless exceptions are disabled, violations of memory alignment cause an alignment exception. Some instructions-for example, many of the Video ALU instructions-automatically disable alignment exceptions because the data may not be properly aligned when stored in memory. Alignment exceptions may be disabled by issuing the `DISALGNEXCPT` instruction in parallel with a load/store operation.

Normally, the memory system requires two address alignments:

- 32-bit word load/stores are accessed on four-byte boundaries, meaning the two least significant bits of the address are b#00.
- 16-bit word load/stores are accessed on two-byte boundaries, meaning the least significant bit of the address must be b#0.

The **Addressing Modes, Transfers, and Sizes** table summarizes the types of transfers and transfer sizes supported by the addressing modes.

Table 6-1: Addressing Modes, Transfers, and Sizes

Addressing Mode	Types of Transfers Supported	Transfer Sizes
Auto-increment Auto-decrement Indirect Indexed Direct	To and from Data Registers	LOADS: 32-bit word 16-bit, zero extended half word 16-bit, sign extended half word 8-bit, zero extended byte 8-bit, sign extended byte STORES: 32-bit word 16-bit half word 8-bit byte

Table 6-1: Addressing Modes, Transfers, and Sizes (Continued)

Addressing Mode	Types of Transfers Supported	Transfer Sizes
	To and from Pointer Registers	LOAD: 32-bit word STORE: 32-bit word
Post-increment	To and from Data Registers	LOADS: 32-bit word 16-bit half word to Data Register high half 16-bit half word to Data Register low half 16-bit, zero extended half word 16-bit, sign extended half word STORES: 32-bit word 16-bit half word from Data Register high half 16-bit half word from Data Register low half

ATTENTION: Be careful when using the `DISALGNEXCPT` instruction, because it disables automatic detection of memory alignment errors. The `DISALGNEXCPT` instruction only affects misaligned loads that use index register indirect addressing. Misaligned loads using pointer register addressing will still cause an exception.

The **Addressing Modes** table summarizes the addressing modes. In the table, an asterisk (*) indicates the processor supports the addressing mode.

Table 6-2: Addressing Modes

	32-bit word	16-bit half-word	8-bit byte	Sign/zero extend	Data Register	Pointer register	Data Register Half
P Auto-inc [P0++]	*	*	*	*	*	*	
P Auto-dec [P0--]	*	*	*	*	*	*	
P Indirect [P0]	*	*	*	*	*	*	*
P Indexed [P0+im]	*	*	*	*	*	*	
FP indexed [FP+im]	*				*	*	
P Post-inc [P0++P1]	*	*		*	*		*
I Auto-inc [I0++]	*	*			*		*

Table 6-2: Addressing Modes (Continued)

	32-bit word	16-bit half-word	8-bit byte	Sign/zero extend	Data Register	Pointer register	Data Register Half
I Auto-dec [I0--]	*	*			*		*
I Indirect [I0]	*	*			*		*
I Post-inc [I0++M0]	*				*		
Direct [im]	*	*	*	*	*	*	*

Addressing Mode Summary

The **Addressing Modes, Transfers, and Sizes** table summarizes the types of transfers and transfer sizes supported by the addressing modes.

Table 6-3: Addressing Modes, Transfers, and Sizes

Addressing Mode	Types of Transfers Supported	Transfer Sizes
Auto-increment Auto-decrement Indirect Indexed Direct	To and from Data Registers	LOADS: 32-bit word 16-bit, zero extended half word 16-bit, sign extended half word 8-bit, zero extended byte 8-bit, sign extended byte STORES: 32-bit word 16-bit half word 8-bit byte
	To and from Pointer Registers	LOAD: 32-bit word STORE: 32-bit word
Post-increment	To and from Data Registers	LOADS: 32-bit word 16-bit half word to Data Register high half 16-bit half word to Data Register low half 16-bit, zero extended half word 16-bit, sign extended half word STORES: 32-bit word 16-bit half word from Data Register high half 16-bit half word from Data Register low half

The **Addressing Modes** table summarizes the addressing modes. In the table, an asterisk (*) indicates the processor supports the addressing mode.

Table 6-4: Addressing Modes

	32-bit word	16-bit half-word	8-bit byte	Sign/zero extend	Data Register	Pointer register	Data Register Half
P Auto-inc [P0++]	*	*	*	*	*	*	
P Auto-dec [P0--]	*	*	*	*	*	*	
P Indirect [P0]	*	*	*	*	*	*	*
P Indexed [P0+im]	*	*	*	*	*	*	
FP indexed [FP+im]	*				*	*	
P Post-inc [P0++P1]	*	*		*	*		*
I Auto-inc [I0++]	*	*			*		*
I Auto-dec [I0--]	*	*			*		*
I Indirect [I0]	*	*			*		*
I Post-inc [I0++M0]	*				*		
Direct [im]	*	*	*	*	*	*	*

AAU Instruction Summary

The **AAU Instructions** table lists the AAU instructions. In the table, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- Preg denotes any Pointer register, `FP`, or `SP` register.
- Ireg denotes any Index register.
- Mreg denotes any Modify register.

- W denotes a 16-bit wide value.
- B denotes an 8-bit wide value.
- immA denotes a signed, A-bits wide, immediate value.
- uimmAmB denotes an unsigned, A-bits wide, immediate value that is an even multiple of B.
- Z denotes the zero-extension qualifier.
- X denotes the sign-extension qualifier.
- BREV denotes the bit-reversal qualifier.

AAU instructions do not affect the ASTAT status bits.

Table 6-5: AAU Instructions

Instruction
Preg = [Preg] ;
Preg = [Preg ++] ;
Preg = [Preg --] ;
Preg = [Preg + uimm6m4] ;
Preg = [Preg + uimm17m4] ;
Preg = [Preg - uimm17m4] ;
Preg = [FP - uimm7m4] ;
Dreg = [Preg] ;
Dreg = [Preg ++] ;
Dreg = [Preg --] ;
Dreg = [Preg + uimm6m4] ;
Dreg = [Preg + uimm17m4] ;
Dreg = [Preg - uimm17m4] ;
Dreg = [Preg ++ Preg] ;
Dreg = [FP - uimm7m4] ;
Dreg = [uimm32] ;
Dreg = [Ireg] ;
Dreg = [Ireg ++] ;
Dreg = [Ireg --] ;
Dreg = [Ireg ++ Mreg] ;
Dreg =W [Preg] (Z) ;
Dreg =W [Preg ++] (Z) ;

Table 6-5: AAU Instructions (Continued)

Instruction
Dreg =W [Preg --] (Z) ;
Dreg =W [Preg + uimm5m2] (Z) ;
Dreg =W [Preg + uimm16m2] (Z) ;
Dreg =W [Preg - uimm16m2] (Z) ;
Dreg =W [Preg ++ Preg] (Z) ;
Dreg =W [uimm32] (Z) ;
Dreg = W [Preg] (X) ;
Dreg = W [Preg ++] (X) ;
Dreg = W [Preg --] (X) ;
Dreg =W [Preg + uimm5m2] (X) ;
Dreg =W [Preg + uimm16m2] (X) ;
Dreg =W [Preg - uimm16m2] (X) ;
Dreg =W [Preg ++ Preg] (X) ;
Dreg =W [uimm32] (X) ;
Dreg_hi = W [Ireg] ;
Dreg_hi = W [Ireg ++] ;
Dreg_hi = W [Ireg --] ;
Dreg_hi = W [Preg] ;
Dreg_hi = W [Preg ++ Preg] ;
Dreg_hi = W [uimm32] ;
Dreg_lo = W [Ireg] ;
Dreg_lo = W [Ireg ++] ;
Dreg_lo = W [Ireg --] ;
Dreg_lo = W [Preg] ;
Dreg_lo = W [Preg ++ Preg] ;
Dreg_lo = W [uimm32] ;
Dreg = B [Preg] (Z) ;
Dreg = B [Preg ++] (Z) ;
Dreg = B [Preg --] (Z) ;
Dreg = B [Preg + uimm15] (Z) ;
Dreg = B [Preg - uimm15] (Z) ;
Dreg = B [uimm32] (Z) ;
Dreg = B [Preg] (X) ;

Table 6-5: AAU Instructions (Continued)

Instruction
Dreg = B [Preg ++] (X) ;
Dreg = B [Preg --] (X) ;
Dreg = B [Preg + uimm15] (X) ;
Dreg = B [Preg - uimm15] (X) ;
Dreg = B [uimm32] (X) ;
[Preg] = Preg ;
[Preg ++] = Preg ;
[Preg --] = Preg ;
[Preg + uimm6m4] = Preg ;
[Preg + uimm17m4] = Preg ;
[Preg - uimm17m4] = Preg ;
[FP - uimm7m4] = Preg ;
[uimm32] = Preg ;
[Preg] = Dreg ;
[Preg ++] = Dreg ;
[Preg --] = Dreg ;
[Preg + uimm6m4] = Dreg ;
[Preg + uimm17m4] = Dreg ;
[Preg - uimm17m4] = Dreg ;
[Preg ++ Preg] = Dreg ;
[FP - uimm7m4] = Dreg ;
[uimm32] = Dreg ;
[Ireg] = Dreg ;
[Ireg ++] = Dreg ;
[Ireg --] = Dreg ;
[Ireg ++ Mreg] = Dreg ;
W [Ireg] = Dreg_hi ;
W [Ireg ++] = Dreg_hi ;
W [Ireg --] = Dreg_hi ;
W [Preg] = Dreg_hi ;
W [Preg ++ Preg] = Dreg_hi ;
W [uimm32] = Dreg_hi ;
W [Ireg] = Dreg_lo ;

Table 6-5: AAU Instructions (Continued)

Instruction
W [Ireg ++] = Dreg_lo ;
W [Ireg --] = Dreg_lo ;
W [Preg] = Dreg_lo ;
W [Preg ++ Preg] = Dreg_lo ;
W [uimm32] = Dreg_lo ;
W [Preg] = Dreg ;
W [Preg ++] = Dreg ;
W [Preg --] = Dreg ;
W [Preg + uimm5m2] = Dreg ;
W [Preg + uimm16m2] = Dreg ;
W [Preg - uimm16m2] = Dreg ;
W [uimm32] = Dreg ;
B [Preg] = Dreg ;
B [Preg ++] = Dreg ;
B [Preg --] = Dreg ;
B [Preg + uimm15] = Dreg ;
B [Preg - uimm15] = Dreg ;
B [uimm32] = Dreg ;
Preg = imm7 (X) ;
Preg = imm16 (X) ;
Preg = uimm32;
Preg += Preg (BREV) ;
Ireg += Mreg (BREV) ;
Preg = Preg << 2 ;
Preg = Preg >> 2 ;
Preg = Preg >> 1 ;
Preg = Preg + Preg << 1 ;
Preg = Preg + Preg << 2 ;
Preg -= Preg ;
Ireg -= Mreg ;

ADSP-BF70x Address Arithmetic Unit (REGFILE) Register Descriptions

Register File (REGFILE) contains the following registers.

Table 6-6: ADSP-BF70x REGFILE Register List

Name	Description
SP	Frame Pointer Register
FP	Stack Pointer Register
USP	User Stack Pointer Register
Pn	Pointer Register
In	Index (Circular Buffer) Register
Mn	Modify (Circular Buffer) Register
Bn	Base (Circular Buffer) Register
Ln	Length (Circular Buffer) Register

Pointer Register

The Pn registers (pointer register files) are 32 bits wide. Although pointer registers are primarily used for address calculations, these registers may also be used for general integer arithmetic with a limited set of arithmetic operations. For instance, pointer register math may be used to maintain counters. Unlike the data registers (Rn, pointer register arithmetic does not affect the arithmetic status (ASTAT) register status bits.

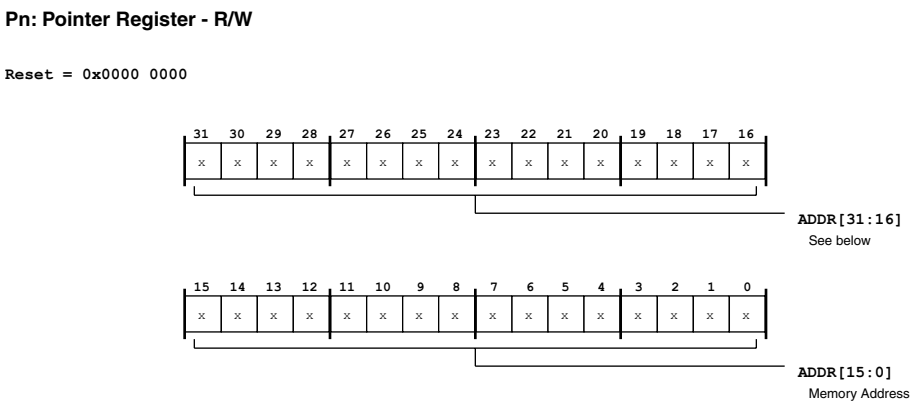


Figure 6-4: Pn Register Diagram

Table 6-7: Pn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Memory Address. The Pn.ADDR bit field (bits 0:31) holds either an address for address calculations or data for arithmetic operations.

Frame Pointer Register

In many respects, the stack and frame pointer registers perform like the other Pn registers. The SP and FP registers may act as general pointers in any of the load/store instructions. But, the SP and FP registers have additional functionality. To speed up context switching, there are two stack pointer registers, a user stack pointer (USP) and a supervisor stack pointer (SP). In assembly code syntax, one should always just use the SP syntax. Based on the current processor mode (supervisor or user), the correct register stack pointer will be used. For context switching, in supervisor mode, one may explicitly refer to the user stack pointer by using the USP syntax. Trying to use USP syntax in user mode causes an exception. Some load and store instructions use SP and FP implicitly. These instructions include LINK and UNLINK instructions, which control stack frame space and manage the SP register.

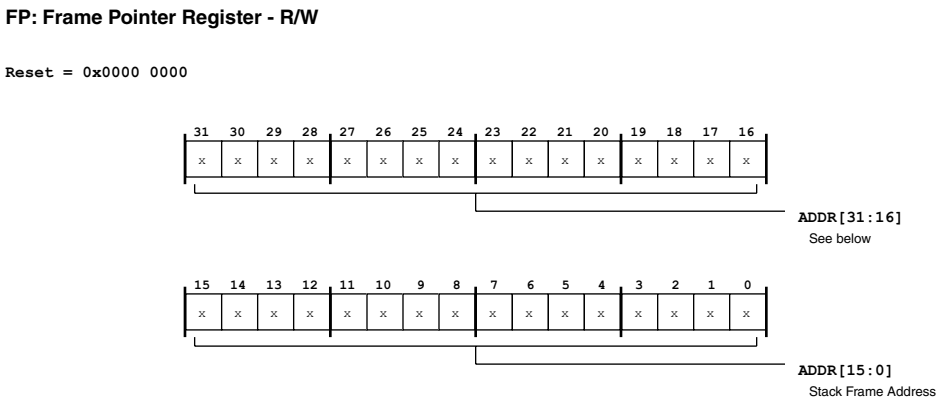


Figure 6-5: FP Register Diagram

Table 6-8: FP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Stack Frame Address. The FP.ADDR bit field (bits 0:31) hold either an address for address calculations or stack operations.

Stack Pointer Register

In many respects, the stack and frame pointer registers perform like the other Pn registers. The SP and FP registers may act as general pointers in any of the load/store instructions. But, the SP and FP registers have additional functionality. To speed up context switching, there are two stack pointer registers, a user stack pointer (USP) and a supervisor stack pointer (SP). In assembly code syntax, one should always just use the SP syntax. Based on the current processor mode (supervisor or user), the correct register stack pointer will be used. For context switching, in supervisor mode, one may explicitly refer to the user stack pointer by using the USP syntax. Trying to use USP syntax in user mode causes an exception. Some load and store instructions use SP and FP implicitly. These instructions include LINK and UNLINK instructions, which control stack frame space and manage the SP register.

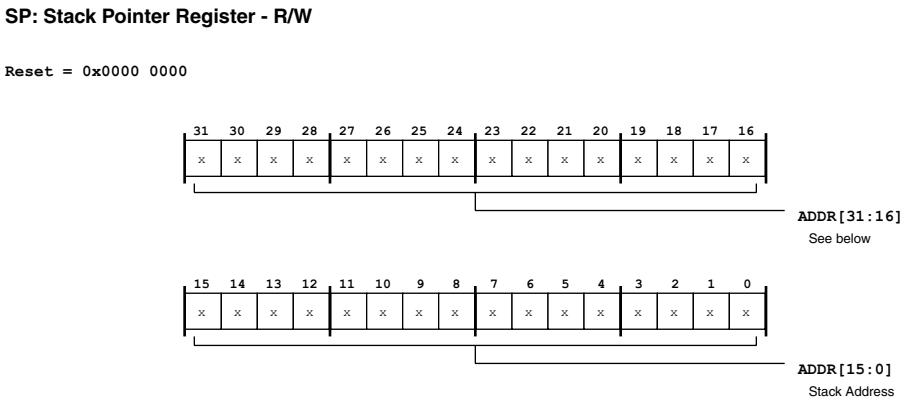


Figure 6-6: SP Register Diagram

Table 6-9: SP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Stack Address. The SP.ADDR bit field (bits 0:31) holds an address (pointer) for address calculations or stack operations.

User Stack Pointer Register

In many respects, the stack and frame pointer registers perform like the other Pn registers. The SP and FP registers may act as general pointers in any of the load/store instructions. But, the SP and FP registers have additional functionality. To speed up context switching, there are two stack pointer registers, a user stack pointer (USP) and a supervisor stack pointer (SP). In assembly code syntax, one should always just use the SP syntax. Based on the current processor mode (supervisor or user), the correct register stack pointer will be used. For context switching, in supervisor mode, one may explicitly refer to the user stack pointer by using the USP syntax. Trying to use USP syntax in user mode causes an exception. Some load and store instructions use SP and FP implicitly. These instructions include LINK and UNLINK instructions, which control stack frame space and manage the SP register.

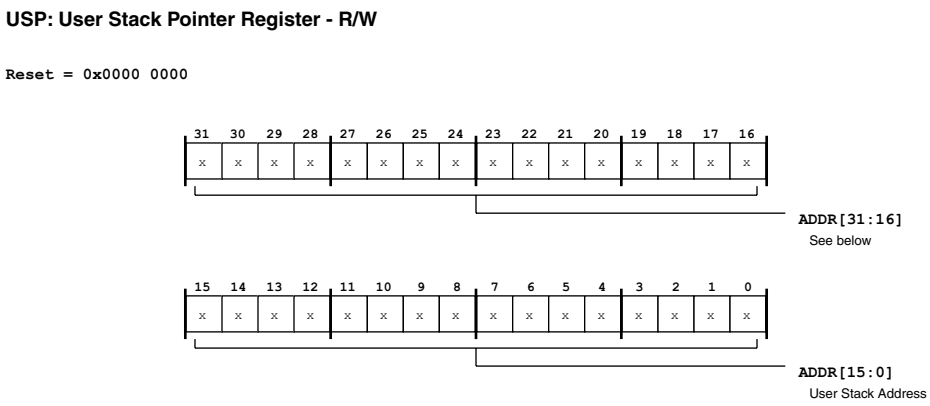


Figure 6-7: USP Register Diagram

Table 6-10: USP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	User Stack Address. The USP.ADDR bit field (bits 0:31) holds an address (pointer) for address calculations or stack operations in user mode.

Index (Circular Buffer) Register

DSP instructions primarily use the data address generator (DAG) register set for addressing. The data address register set consists of these registers:

- I3 through I0 contain index addresses
- M3 through M0 contain modify values
- B3 through B0 contain base addresses
- L3 through L0 contain length values

All data address registers are 32 bits wide.

The I (index) registers and B (base) registers always contain addresses of 8-bit bytes in memory. The index registers contain an effective address. The M (modify) registers contain an offset value that is added to one of the index registers or subtracted from it.

The B and L (length) registers define circular buffers. The B register contains the starting address of a buffer, and the L register contains the length in bytes. Each L and B register pair is associated with the corresponding I register. For example, L0 and B0 are always associated with I0. However, any M register may be associated with any I register. For example, I0 may be modified by M3.

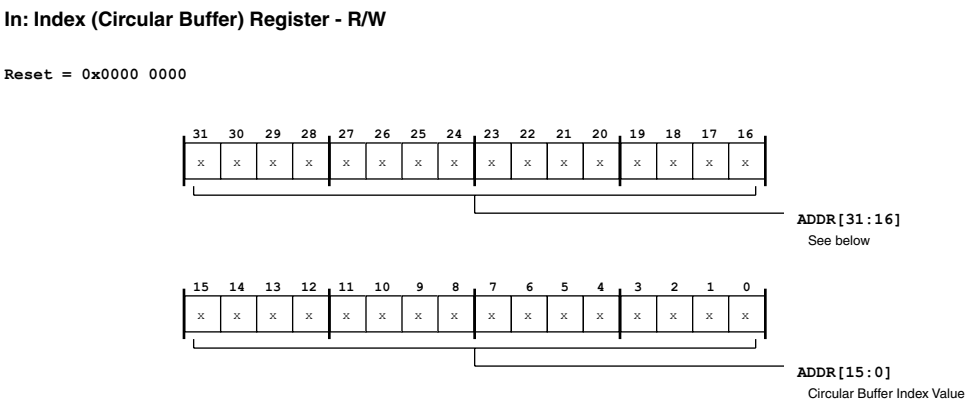


Figure 6-8: In Register Diagram

Table 6-11: In Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Circular Buffer Index Value. The `In.ADDR` bits contain the circular buffer index address value.

Modify (Circular Buffer) Register

DSP instructions primarily use the data address generator (DAG) register set for addressing. The data address register set consists of these registers:

- `I3` through `I0` contain index addresses
- `M3` through `M0` contain modify values
- `B3` through `B0` contain base addresses
- `L3` through `L0` contain length values

All data address registers are 32 bits wide.

The I (index) registers and B (base) registers always contain addresses of 8-bit bytes in memory. The index registers contain an effective address. The M (modify) registers contain an offset value that is added to one of the index registers or subtracted from it.

The B and L (length) registers define circular buffers. The B register contains the starting address of a buffer, and the L register contains the length in bytes. Each L and B register pair is associated with the corresponding I register. For example, `L0` and `B0` are always associated with `I0`. However, any M register may be associated with any I register. For example, `I0` may be modified by `M3`.

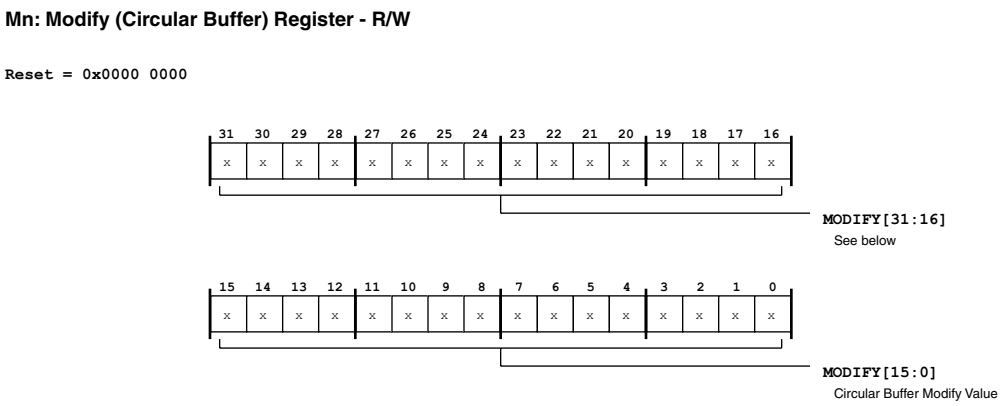


Figure 6-9: Mn Register Diagram

Table 6-12: Mn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	MODIFY	Circular Buffer Modify Value. The `Mn.MODIFY` bits contain the circular buffer modify value.

Base (Circular Buffer) Register

DSP instructions primarily use the data address generator (DAG) register set for addressing. The data address register set consists of these registers:

- `I3` through `I0` contain index addresses
- `M3` through `M0` contain modify values
- `B3` through `B0` contain base addresses
- `L3` through `L0` contain length values

All data address registers are 32 bits wide.

The I (index) registers and B (base) registers always contain addresses of 8-bit bytes in memory. The index registers contain an effective address. The M (modify) registers contain an offset value that is added to one of the index registers or subtracted from it.

The B and L (length) registers define circular buffers. The B register contains the starting address of a buffer, and the L register contains the length in bytes. Each L and B register pair is associated with the corresponding I register. For example, `L0` and `B0` are always associated with `I0`. However, any M register may be associated with any I register. For example, `I0` may be modified by `M3`.

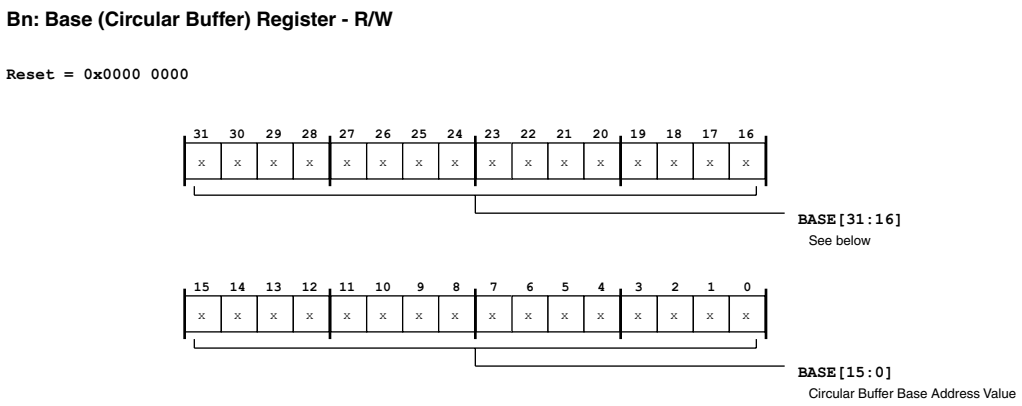


Figure 6-10: Bn Register Diagram

Table 6-13: Bn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	BASE	Circular Buffer Base Address Value. The Bn.BASE bits contain the circular buffer base address value.

Length (Circular Buffer) Register

DSP instructions primarily use the data address generator (DAG) register set for addressing. The data address register set consists of these registers:

- I3 through I0 contain index addresses
- M3 through M0 contain modify values
- B3 through B0 contain base addresses
- L3 through L0 contain length values

All data address registers are 32 bits wide.

The I (index) registers and B (base) registers always contain addresses of 8-bit bytes in memory. The index registers contain an effective address. The M (modify) registers contain an offset value that is added to one of the index registers or subtracted from it.

The B and L (length) registers define circular buffers. The B register contains the starting address of a buffer, and the L register contains the length in bytes. Each L and B register pair is associated with the corresponding I register. For example, L0 and B0 are always associated with I0. However, any M register may be associated with any I register. For example, I0 may be modified by M3.

Ln: Length (Circular Buffer) Register - R/W

Reset = 0x0000 0000

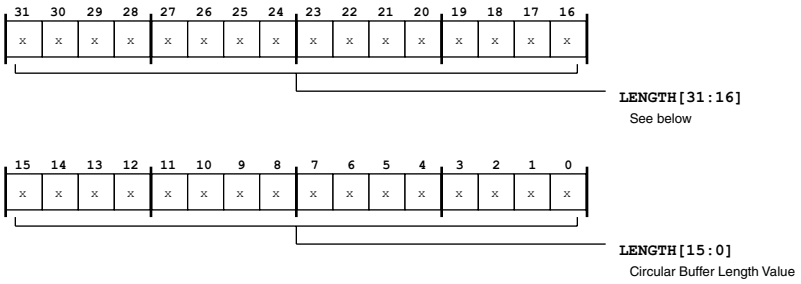


Figure 6-11: Ln Register Diagram

Table 6-14: Ln Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	LENGTH	Circular Buffer Length Value. The Ln.LENGTH bits contain the circular buffer length value.

7 Memory

Blackfin processors support a hierarchical memory model with different performance and size parameters, depending on the memory location within the hierarchy. Level 1 (L1) memories interconnect closely and efficient with the Blackfin core for best performance. Separate blocks of L1 memory can be accessed simultaneously through multiple bus systems. Instruction memory is separated from data memory, but unlike classical Harvard architectures, all L1 memory blocks are accessed by one unified addressing scheme. Portions of L1 memory can be configured to function as cache memory. Some Blackfin derivatives also feature on-chip Level 2 (L2) memories. L2 memories can freely store instructions and data. L2 memories take more core clock cycles to access than L1 memories. The processors support an external memory space that includes asynchronous memory space for static RAM devices and synchronous memory space for dynamic RAM such as SDRAM devices. This chapter discusses the architecture and principles of L1 memories as well as memory protection and caching mechanisms. For memory size, population, L2, and off-chip memory interfaces, refer to the specific *Blackfin+ Processor Hardware Reference* for your derivative.

Memory Architecture

Blackfin processors have a unified 4G byte address range that spans a combination of on-chip and off-chip memory and memory-mapped I/O resources. Of this range, some of the address space is dedicated to internal, on-chip resources. The processor populates portions of this internal memory space with:

- L1 Static Random Access Memories (SRAM)
- L2 Static Random Access Memories (SRAM)
- A set of memory-mapped registers (MMRs)
- A boot Read-Only Memory (ROM)

The figure shows a processor memory architecture typical of most Blackfin processors.

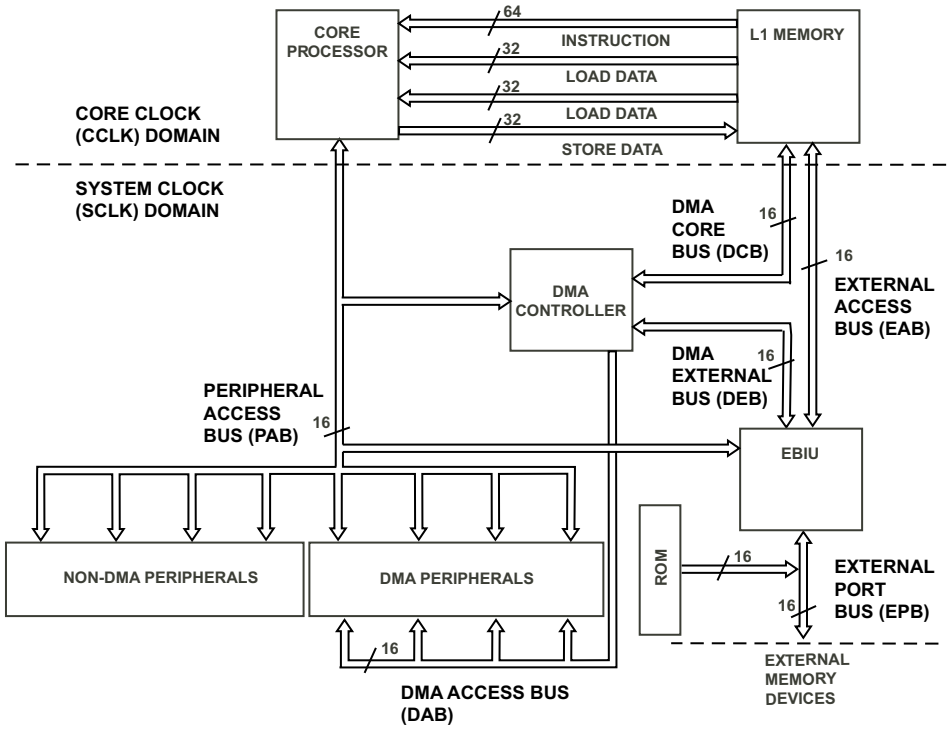


Figure 7-1: Processor Memory Architecture

Overview of On-Chip Level 1 (L1) Memory

The L1 memory system performance provides high bandwidth and low latency. Because SRAMs provide deterministic access time and very high throughput, DSP systems have traditionally achieved performance improvements by providing fast SRAM on the chip.

The addition of instruction and data caches (SRAMs with cache control hardware) provides both high performance and a simple programming model. Caches eliminate the need to explicitly manage data movement into and out of L1 memories. Code can be ported to or developed for the processor quickly without requiring performance optimization for the memory organization.

The figure in Memory Architecture shows the typical bus architecture of single-core Blackfin devices that do not feature L2 memories on-chip. The bus widths on the system side may vary.

The L1 memory provides:

- A modified Harvard architecture, allowing up to four core memory accesses per clock cycle (one 64-bit instruction fetch, two 32-bit data loads, and one pipelined 32-bit data store)
- Simultaneous system DMA, cache maintenance, and core accesses
- SRAM access at processor clock rate (CCLK) for critical DSP algorithms and fast context switching

- Instruction and data cache options for micro controller code, excellent High Level Language (HLL) support, and ease of programming cache control instructions, such as `PREFETCH` and `FLUSH`
- Memory protection

The L1 memories operate at the core clock frequency (CCLK).

L1 Instruction memory is a continuous region of memory which may only be used to store instructions. A portion of this memory may be configured as instruction cache or dedicated instruction SRAM, and the remainder may only be used as instruction SRAM. L1 instruction memory cannot normally be accessed by load or store instructions.

Typical L1 data memory is divided into three blocks. Two, known as block A and block B, are equal sized and may be configured as cache or SRAM, and the third, known as block C or scratchpad, is a small region of dedicated data SRAM often used for system stacks and heaps. Concurrent accesses by the core or DMA, to separate blocks proceed in parallel, whereas concurrent accesses to the same block may be stalled due to a sub-bank conflict.

The L1 memory system contains special purpose memory spaces, such as cache tags and parity bits, to which direct access by load or store instructions is normally restricted. An extended data access mode is supported in which all restricted memory spaces including L1 instruction memory may be directly accessed by suitably privileged software.

Overview of other On-Chip and Off-Chip Memories

Some Blackfin derivatives feature a Level 2 (L2) memory on chip. The L2 memory provides low latency, high-bandwidth capacity. This memory system is referred to as on-chip L2 because it forms an on-chip memory hierarchy with L1 memory. On-chip L2 memory provides more capacity than L1 memory, but the latency is higher. It is capable of storing both instructions and data. In some derivatives may only be configured as cache and on others only as SRAM.

Most Blackfin processors feature an on-chip Boot ROM. Processors also feature external memory controllers which enable access to external DRAM via standard protocols such as DDR2, as well as serial memory and flash. The external memory is sometimes referred to as Level 3 (L3).

For details of memory map, off-chip and on-chip memory, refer to the specific *Blackfin+ Processor Hardware Reference* for your derivative.

L1 Instruction Memory

L1 Instruction Memory is a continuous region of the address space which may only be use for storing instructions. On all Blackfin processors part of this space must be used as directly addresses SRAM. On many Blackfin processors another part of the L1 Instruction Memory may be configured as cache or as

SRAM. Control bits in the L1_IM_ICTL register can be used to organize that part of L1 Instruction Memory as:

- A simple SRAM
- A 4-Way, set associative instruction cache

The L1 Instruction memory content is not accessible by load or store operations in normal operation. This memory's content may be read and modified using DMA and by loads and stores in extended data access mode.

L1_IM_ICTL Register

The L1 Instruction Memory Control register (L1_IM_ICTL) contains control bits for the L1 Instruction Memory. By default after reset, cache and Cacheability Protection Lookaside Buffer (CPLB) address checking is disabled. For more information, see L1 Instruction Cache and the Instruction Memory Control Register.

The CFG bit reserves a portion of L1 instruction SRAM to serve as cache. Note reserving memory to serve as cache will not alone enable L2 and off-chip memory accesses to be cached. CPLBs must also be enabled using the ENCPLB bit and the CPLB descriptors (L1_IM_ICPLB_DFLT or L1_IM_ICPLB_DATAx and L1_IM_ICPLB_ADDRx registers) must specify desired memory pages as cache-enabled.

Instruction CPLBs are disabled by default after reset. When disabled, only minimal address checking is performed by the L1 memory interface. This minimal checking generates an exception to the processor whenever it attempts to fetch an instruction from the following:

- Reserved (non populated) L1 instruction memory space
- L1 data memory space
- MMR space

CPLBs must be disabled using the ENCPLB bit prior to updating their descriptors (L1_IM_ICPLB_DFLT or L1_IM_ICPLB_DATAx and L1_IM_ICPLB_ADDRx registers). Note since load store ordering is weak (see Ordering of Loads and Stores), disabling of CPLBs should be proceeded by a CSYNC.

The RDCHK bit enables read parity error detection in L1 Instruction Memory on Blackfin processors that support this feature. See L1 Parity Protection for more details.

The CBYPASS bit enables cache bypass mode. When the cache is enabled and this bit is set, instructions in cacheable memory are loaded directly from the system without disturbing the contents of the cache. An SSYNC instruction should be executed after setting or clearing this bit to ensure it has taken effect.

When the CPRIORST bit is set to 1, the cached states of all CPRIO bits are cleared. This simultaneously forces all cached lines to be of equal (low) importance. Cache replacement policy is based first on line importance indicated by the cached states of the CPRIO bits, and then on a heuristic replacement algorithm. See Instruction Cache Locking by Line for complete details. This bit must be 0 to allow the state of the CPRIO bits to be stored when new lines are cached.

All mode changes take effect when the control register is written, but due to the processor pipeline the effect may not impact instructions immediately following the write. To ensure consistency, you should issue a `CSYNC` instruction after writing the control register. If a mode change somehow affects the system other than the core or L1, then `SSYNC` should be used instead. For instance if the `RDCHK` bit is set to enable parity checking and instructions are being executed directly out of L1 SRAM the instruction following the write to the control register will have been read from L1 and be in the pipeline at the time parity checking is enabled. If that instruction is a `CSYNC` it will cause the pipeline to be flushed and the subsequent instruction to be fetched from L1 SRAM, parity will be checked and an error detected. If cache is also enabled then an `SSYNC` before the write to the control register will ensure all cache fills are completed and tags updated before parity checking is enabled.

L1 Instruction SRAM

The processor core reads the instruction memory through the 64-bit wide instruction fetch bus. All addresses from this bus are 64-bit aligned. Each instruction fetch can return any combination of 16-, 32- or 64-bit instructions (for example, four 16-bit instructions, two 16-bit instructions and one 32-bit instruction, or one 64-bit instruction).

The pointer registers and index registers, which are described in the *Address Arithmetic Unit* chapter, may only access L1 Instruction Memory directly when extended data access is enabled (see Extended Data Access), otherwise a direct access to an address in instruction memory SRAM space generates an exception. Write access to the L1 Instruction SRAM Memory may also be made through the 64-bit wide system DMA port.

Typically the SRAM is implemented as a collection of single ported subbanks, Writes to one subbank may proceed in parallel with reads from another, making the instruction memory effectively dual ported. For details, refer to the specific *Blackfin+ Processor Hardware Reference* for your derivative.

L1 Instruction Cache

For information about cache terminology, see Terminology.

On some Blackfin derivatives, the L1 Instruction Memory may also be configured to contain a, 16K byte 4-Way set associative instruction cache. To improve the average access latency for critical code sections, each line of the cache can be locked independently. When the memory is configured as cache, it can only be accessed directly by loads and stores if extended data access mode is enabled.

When cache is enabled, only memory pages further specified as cacheable by the CPLBs will be cached. When CPLBs are enabled, any memory location that is accessed must have an associated page definition available, or a CPLB exception is generated. CPLBs are described in Memory Protection and Properties.

The figures in Cache Lines shows the overall Blackfin processor instruction cache organization.

L1 instruction cache is an optional feature of the Blackfin architecture, processors capable of configuring part of L1 instruction SRAM as cache have the ICACHE bit set in the read-only FEATURE0 register.

Cache Lines

As shown in the Instruction Cache Organization figure, the cache consists of a collection of cache lines. Each cache line is made up of a tag component and a data component.

- The tag component incorporates a 20-bit address tag, replacement policy bits, including a Priority bit, and a Valid bit.
- The data component is made up of four 64-bit words of instruction data.

The tag and data components of cache lines are stored in the tag and data memory arrays, respectively.

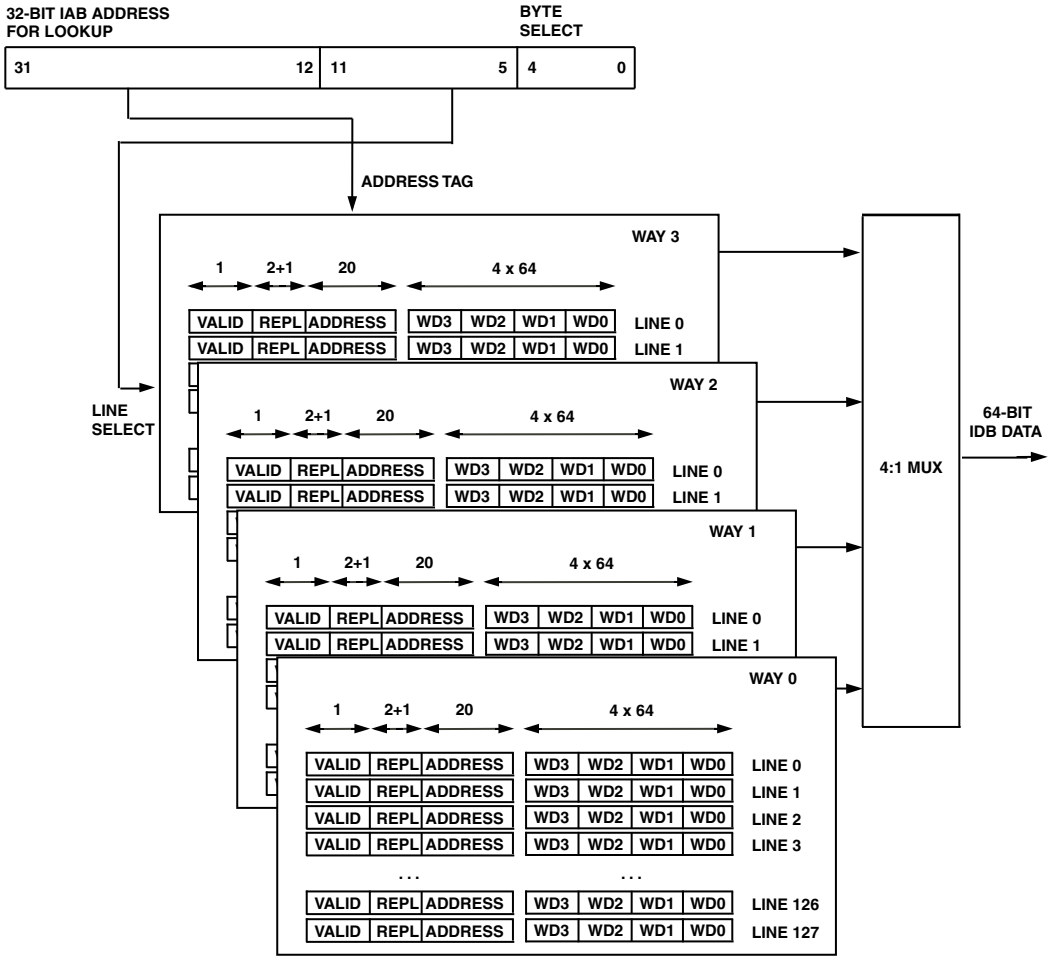


Figure 7-2: Instruction Cache Organization

The Replacement bits are part of a replacement algorithm used to determine which cache line should be replaced if a cache miss occurs.

The Valid bit indicates the state of a cache line. A cache line is always valid or invalid.

- Invalid cache lines have their Valid bit cleared, indicating the line will be ignored during an address-tag compare operation.
- Valid cache lines have their Valid bit set, indicating the line contains valid instruction/data that is consistent with the source memory.

The tag and data components of a cache line are illustrated in the Cache Line - Tag and Data Portions figure.

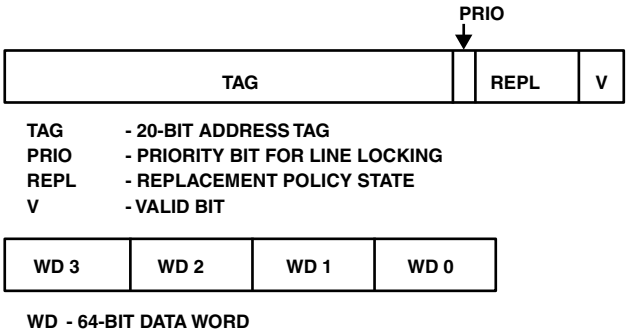


Figure 7-3: Cache Line - Tag and Data Portions

Cache Hits and Misses

A cache hit occurs when the address for an instruction fetch request from the core matches a valid entry in the cache. Specifically, a cache hit is determined by comparing the upper 20 bits of the instruction fetch address to the address tags of valid lines currently stored in a cache set. The cache set (cache line across ways) is selected, using bits 11 through 5 of the instruction fetch address. If the address-tag compare operation results in a match in any of the four ways and the respective cache line is valid, a cache hit occurs. If the address-tag compare operation does not result in a match in any of the four ways or the respective line is not valid, a cache miss occurs.

When a cache miss occurs, the instruction memory unit generates a cache line fill access to retrieve the missing instruction from memory that is external to the core. The address for the external memory access is the address of the 32 byte block containing the target instruction word.

A location in the cache is selected to store the missing block once the cache line fill has completed. If the tag-address compare operation results in a cache miss, the Valid, Replacement and Priority bits for the selected set are examined by a cache line replacement unit to determine the entry to use for the new cache line, that is, whether to use Way0, Way1, Way2, or Way3.

The cache line replacement unit first checks for invalid entries (that is, entries having its Valid bit cleared). If there are no invalid entries in the set, an entry with the Priority bit cleared is selected. In the last resort the cache line replacement unit will select an entry with its Priority bit set. Valid entries are selected using a replacement algorithm designed to choose the entry that is statistically least likely to be accessed again. The Replacement bits play a role in the implementation of this algorithm.

When a cache miss occurs, the core halts until the target instruction word is returned from external memory.

Instruction Cache Management

The system DMA controller and the core DAGs cannot normally access the instruction cache directly. By a combination of instructions and the use of extended data access mode, it is possible to initialize the instruction tag and data arrays and provide a mechanism for instruction cache test, initialization, and debug.

See Extended Data Access.

Instruction Cache Locking by Line

The `CPRIO` bits in the `L1_IM_ICPLB_DATAx` registers (see Memory Protection and Properties) are used to enhance control over which code remains resident in the instruction cache. When a cache line is filled, the state of this bit is stored along with the line's tag. It is then used in conjunction with the replacement policy to determine which Way is victimized if all cache Ways are occupied when a new cacheable line is fetched. This bit indicates that a line is of either "low" or "high" importance. In a modified replacement policy, a high can replace a low, but a low cannot replace a high. If all Ways are occupied by highs a cacheable low may replace a high. If all previously cached highs ever become less important, they may be simultaneously transformed into lows by writing to the `CPRIORST` bit in the `L1_IM_ICTL` register.

Instruction Cache Invalidation

The instruction cache can be invalidated as a whole, by address, or by cache line.

The simplest method for invalidating the complete instruction cache is to disable and then re-enable the cache. By clearing the `CFG` bit in the `L1_IM_ICTL` register, all Valid bits in the instruction cache are set to the invalid state. A second write to the `L1_IM_CTL` register to set the `CFG` bit configures the instruction memory as cache again. An `SSYNC` instruction should be run before invalidating the cache and a `CSYNC` instruction should be inserted after each of these operations.

The `IFLUSH` instruction can explicitly invalidate cache lines based on their line addresses. The target address of the instruction is generated from the pointer registers. Because the instruction cache should not contain modified (dirty) data, the cache line is simply invalidated, and not "flushed."

In the following example, the `P2` register contains the address of a valid memory location. If this address has been brought into cache, the corresponding cache line is invalidated after the execution of this instruction.

Example of `IFLUSH` instruction:

```
IFLUSH [ P2 ] ;   /* Invalidate cache line containing address that P2 points to */
```

Because the `IFLUSH` instruction is used to invalidate a specific address in the memory map and its corresponding cache-line, it is most useful when the buffer being invalidated is less than the cache size. For more information about the `IFLUSH` instruction.

Finally larger portions of the cache may be invalidated by writing directly to the Valid bits while extended data access is enabled. (SeeExtended Data Access.)

L1 Data Memory

L1 data memory is organized as a number of distinct blocks, each of which constitutes a separate contiguous region of the address space. Accesses to different blocks are guaranteed not to collide. Accesses within a single block may not collide if they are to different subbanks. When there are no collisions, this L1 data traffic could occur in a single core clock cycle:

- Two 32-bit data loads
- One pipelined 32-bit data store
- One DMA I/O, up to 64 bits
- One 64-bit cache fill/victim access

Typically there are three blocks, A, B and C. Parts of the two larger blocks, A and B, may be configured as data cache. Control bits in the `L1_DM_DCTL` can be used to configure L1 data memory.

The processor cannot fetch instructions directly from L1 data memory.

L1_DM_DCTL Register

The Data Memory Control register (`L1_DM_DCTL`) contains control bits for the L1 Data Memory. For more information, see the Data Memory Control Register.

The `ENDCPLB` bit is used to enable/disable the 16 Cacheability Protection Lookaside Buffers (CPLBs) used for data (see L1 Data Cache). Data CPLBs are disabled by default after reset. When disabled, only minimal address checking is performed by the L1 memory interface. This minimal checking generates an exception when the processor:

- Addresses nonexistent (reserved) L1 memory space
- Attempts to perform a nonaligned memory access
- Attempts to access MMR space either using DAG1 or when in User mode

CPLBs must be disabled using this bit prior to updating their descriptors (registers `L1_DM_DCPLB_DFLT` or `L1_DM_DCPLB_DATAx` and `L1_DM_DCPLB_ADDRx`). Note that since load store ordering is weak (see Ordering of Loads and Stores), disabling CPLBs should be preceded by a `CSYNC` instruction.

By default after reset, all L1 Data Memory serves as SRAM. The `CFG[1:0]` bits can be used to reserve portions of this memory to serve as cache instead. Part of block A may be configured as cache, or parts of both block A and B may be configured as cache.

Reserving memory to serve as cache does not enable non-L1 memory accesses to be cached. To do this, CPLBs must also be enabled (using the ENDCPLB bit) and CPLB descriptors (registers L1_DM_DCPLB_DFLT or L1_DM_DCPLB_DATAx and L1_DM_DCPLB_ADDRx) must specify chosen memory pages as cacheable.

The DCBS bit provides some control over how data structures outside L1 map to the caches in the two data blocks. When both Data Block A and Data Block B have memory serving as cache, the DCBS bit selects either address bit 14 or 23 to steer traffic between the cache sets in each block. When enabled, alternating 8M byte regions are cached in Block A or Block B so are guaranteed not to collide when accessed by the two DAGs in parallel. In the default mode the two caches behave more like a single unified cache which is the more efficient operation for general purpose activity. This bit provides some control over which addresses alias into the same set, so it may also be used to affect which addresses tend to remain resident in cache by avoiding victimization of repetitively used sets. This bit has no effect unless both Data Block A and Data Block B are serving as cache (bits CFG[1:0] in this register are set to 11).

The RDCHK bit enables read parity error detection in L1 Data Memory on Blackfin processors that support this feature. See L1 Parity Protection for more details.

The CBYPASS bit enables cache bypass mode. When the cache is enabled and this bit is set, data in cacheable memory is loaded directly from the system without disturbing the contents of the cache. An SSYNC instruction should be executed after setting or clearing this bit to ensure it has taken effect.

The ENX bit enables extended data access mode, in which access to special purpose L1 memory spaces is permitted. When this mode is enabled the addresses of loads and stores to these restricted regions, including L1 Instruction SRAM, are checked according to the configuration of the Data CPLBs.

All mode changes take effect when the control register is written. To ensure consistency, you should issue a CSYNC instruction after writing the control register. If a mode change somehow affects the system other than the core or L1, then SSYNC should be used instead.

L1 Data SRAM

L1 data SRAM is directly addressable by the processor and by DMA. Data in L1 data SRAM can be accessed by the core with no stalls so long as access collisions are avoided. Therefore configuring the whole of L1 data memory as SRAM is potentially the most efficient way to use it. However this is at the cost of additional software complexity when data structures are larger than available L1 memory. In a common use-case DMA engines transfer data between L1 and the system while the processor processes data previously fetched to L1.

The programmer should ensure stalls due to collisions are infrequent. The Blackfin architecture guarantees that accesses to different blocks do not collide. So, a tight loop which loads two operands per cycle will not stall, due to the operand loads, if the operands can be placed in separate blocks.

Two accesses to the same block may not collide depending upon the memory microarchitecture, which may change between generations of Blackfin processor. However a collision due to two loads in a parallel issue instruction will not take more cycles than the additional cycles due to executing one of the loads in a separate instruction. So a good heuristic is to place loads to the same bank in different instructions unless that requires increasing the number of instructions.

Collisions of DMA and processor accesses are less common because DMA runs at system clock speeds which are some fraction of the core clock. DMA accesses are also usually delayed behind processor accesses but after some delay a fairness algorithm will ensure DMA gets a chance to access L1. So if possible DMA accesses should be to a different block than concurrent accesses by the processor.

L1 Data Cache

For definitions of cache terminology, see Terminology.

Unlike instruction cache, which is 4-Way set associative, data cache is 2-Way set associative. When two blocks are available and enabled as cache, additional sets rather than ways are created. When both Data Block A and Data Block B have memory serving as cache, the DCBS bit in the L1_DM_DCTL register may be used to control whether the sets in each block are to act as a single cache or as two caches which may be accessed in parallel. For best general purpose operation single cache (DCBS=0) should be selected. This provides some control over which addresses alias into the same set, so may also be used to affect which addresses tend to remain resident in cache by avoiding victimization of repetitively used sets.

Two different cache modes are available.

- Write-through cache
- Write-back cache

Cache mode is selected by the data CPLB descriptors (see Memory Protection and Properties). These cache modes can be used simultaneously since cache mode is selectable for each memory page independently.

If cache is enabled (controlled by bits CFG[1:0] in the L1_DM_CTL register), data CPLBs should also be enabled (controlled by ENDCPLB bit in the L1_DM_CTL register). Only memory pages specified as cacheable by data CPLBs will be cached. The default behavior when data CPLBs are disabled is for nothing to be cached.

Access to core MMR space is not controlled by the data CPLBs so cannot be configured as cacheable.

Data cache is an optional feature of the Blackfin architecture. Processors that support a cache in Block A have DCACHE1 set in the FEATURE0 register and processors that support a cache in both blocks A and B have DCACHE2 set in the FEATURE0 register.

Example of Mapping Cacheable Address Space

An example of how the cacheable address space maps into two data blocks follows.

When both blocks are configured as cache they operate as two independent, 16K byte, 2-Way set associative caches that can be independently mapped into the Blackfin processor address space.

If both data blocks are configured as cache, the DCBS bit in the L1_DM_DCTL register designates Address bit A[14] or A[23] as the cache selector. Address bit A[14] or A[23] selects the cache implemented by Data Block A or the cache implemented by Data Block B.

- If DCBS = 0, then A[14] is part of the address index, and all addresses in which A[14] = 0 use Data Block B. All addresses in which A[14] = 1 use Data Block A. In this case, A[23] is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.
- If DCBS = 1, then A[23] is part of the address index, and all addresses where A[23] = 0 use Data Block B. All addresses where A[23] = 1 use Data Block A. In this case, A[14] is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.

The result of choosing DCBS = 0 or DCBS = 1 is:

- If DCBS = 0, A[14] selects Data Block A instead of Data Block B. Alternating 16K byte regions of memory map into each of the two 16K byte caches implemented by the two data blocks. Consequently: As a result, the cache operates as if it were a single, contiguous, 2-Way set associative 32K byte cache. Each Way is 16K byte long, and all data elements with the same first 14 bits of address index to a unique set in which up to two elements can be stored (one in each Way).
  - Any data in the first 16K byte of memory could be stored only in Data Block B. Any data in the next address range (16K byte through 32K byte) - 1 could be stored only in Data Block A. Any data in the next range (32K byte through 48K byte) - 1 would be stored in Data Block B. Alternate mapping would continue.
- If DCBS = 1, A[23] selects Data Block A instead of Data Block B. With DCBS = 1, the system functions more like two independent caches, each a 2-Way set associative 16K byte cache. Each Block serves an alternating set of 8M byte regions of memory. For example, Data Block B caches all data accesses for the first 8M byte of memory address range. That is, every 8M byte of range vies for the two line entries (rather than every 16K byte repeat). Likewise, Data Block A caches data located above 8M byte and below 16M byte.

For example, if DCBS = 1 and the application is working from a data set that is 1M byte long and located entirely in the first 8M byte of memory, it is effectively served by only half the cache, that is, by Data Block B (a 2-Way set associative 16K byte cache). In this instance, the application never derives any benefit from Data Block A.

However, if the application is working from two data sets, located in two memory spaces at least 8M byte apart, closer control over how the cache maps to the data is possible. For example, if the program is doing a series of dual MAC operations in which both DAGs are accessing data on every cycle, by placing DAG0's data set in one 8M byte region of memory and DAG1's data set in the other, the system can ensure that:

- DAG0 gets its data from Data Block A for all of its accesses and
- DAG1 gets its data from Data Block B.

This arrangement causes the core to use both data buses for cache line transfer and achieves the maximum data bandwidth between the cache and the core.

The Data Cache Mapping figure shows an example of how mapping is performed when `DCBS = 1`.

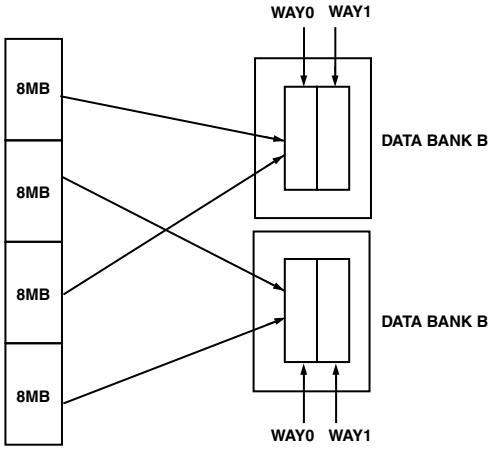


Figure 7-4: Data Cache Mapping When DCBS = 1

Data Cache Access

The Cache Controller tests the address from the DAGs against the tag bits. If the logical address is present in L1 cache, a cache hit occurs, and the data is accessed in L1. If the logical address is not present, a cache miss occurs, and the memory transaction is passed to the next level of memory via the system interface.

The set index and replacement policy for the Cache Controller determines the cache tag and data space that are allocated for the data coming back from external memory.

A data cache line is in one of three states: invalid, exclusive (valid and clean), and modified (valid and dirty). If valid data already occupies the allocated line and the cache is configured for write-back storage, the controller checks the state of the cache line and treats it accordingly:

- If the state of the line is exclusive (clean), the new tag and data write over the old line.
- If the state of the line is modified (dirty), then the cache contains the only valid copy of the data. This is copied back to external memory before the new tag and data is written to the cache.

Cache Write Method

Cache write memory operations can be implemented by using either a write-through method or a write-back method:

- For each store operation, write-through caches initiate a write to external memory immediately upon the write to cache. If the cache line is replaced or explicitly flushed by software, the contents of the cache line are invalidated rather than written back to external memory.
- A write-back cache does not write to external memory until the line is replaced by a load operation that needs the line. For most applications, a write-back cache is more efficient than a write-through cache, as the external memory accesses are less frequent.

Data Cache Control Instructions

The processor defines three data cache control instructions that are accessible in User and Supervisor modes. The instructions are `PREFETCH`, `FLUSH`, and `FLUSHINV`.

- `PREFETCH` (Data Cache Prefetch) attempts to allocate a line into the L1 cache. If the prefetch hits in the cache, generates an exception, or addresses a cache inhibited region, `PREFETCH` functions like a `NOP`. It can be used to begin a data fetch prior to when the processor needs the data, to improve performance.
- `FLUSH` (Data Cache Flush) causes the data cache to synchronize the specified cache line with external memory. If the cached data line is dirty, the instruction writes the line out and marks the line clean in the data cache. If the specified data cache line is already clean or does not exist, `FLUSH` functions like a `NOP`.
- `FLUSHINV` (Data Cache Line Flush and Invalidate) causes the data cache to perform the same function as the `FLUSH` instruction and then invalidate the specified line in the cache. If the line is in the cache and dirty, the cache line is written out to external memory. The Valid bit in the cache line is then cleared. If the line is not in the cache, `FLUSHINV` functions like a `NOP`.

If software requires synchronization with system hardware, place an `SSYNC` instruction after the `FLUSH` instruction to ensure that the flush operation has completed. If ordering is desired to ensure that previous stores have been pushed through all the queues, place an `SSYNC` instruction before the `FLUSH`.

Data Cache Invalidation

Besides the `FLUSHINV` instruction, explained in the previous section, two additional methods are available to invalidate the data cache when flushing is not required.

To invalidate the complete data cache clear the `CFG[1:0]` bits in the `L1_DM_DCTL` register. All Valid bits in the data cache are set to the invalid state. A second write to the `L1_DM_DCTL` register to set the `CFG[1:0]` bits to their previous state then configures the data memory back to its previous cache/SRAM configuration. An `SSYNC` instruction should be run before invalidating the cache and a `CSYNC` instruction should be inserted after each of these operations. This method is useful if the data buffer to be invalidated is greater than the size of the cache.

The second technique directly invalidates cache lines by writing directly to the Valid bits while extended data access is enabled. (See Extended Data Access.)

Extended Data Access

The L1 memory system contains special purpose memory spaces to which direct access by load or store instructions is normally restricted. When extended data access are enabled all restricted memory spaces including L1 instruction memory may be directly accessed by suitably privileged software. The extended data accesses are typically not as fast as regular L1 data SRAM accesses so should only be used for exceptional processing, such as initialization or test.

Setting the `ENX` (Extended Data Access Enable) bit `L1_DM_DCTL` MMR enables data access to restricted L1 memory spaces. When enabled, regular load and store instructions may be used to read or write these memory spaces:

- L1 Data SRAM currently used as cache
- L1 Instruction SRAM whether in use as cache or not

Extended data access is also possible to these additional memory spaces which are mapped into otherwise unused parts of the core's L1 address space.

- Cache Tags
- Cache Dirty Bits
- Parity Bits

When extended data accesses are enabled loads or stores to all the restricted regions, including L1 instruction SRAM, is controlled by the data CPLBs if they are enabled. When extended data accesses are disabled a load or store to any of these restricted memory regions will cause a Data access CPLB protection violation exception irrespective of the CPLB setting.

Loads and stores to restricted memory regions can only be through DAG0. An attempt to access them through DAG1 will cause a Data access CPLB protection violation exception.

Loads and stores to cache tags and dirty bits should be 32-bit wide and 32-bit aligned, or incorrect data may be returned with no exception raised.

Writes to L1 instruction SRAM in order to initialize instruction memory should be followed by a CSYNC instruction to ensure the processor flushes its pipeline and fetches the next instruction from the modified SRAM.

Extended data access mode does not affect DMA access to these memory regions. DMA access is never permitted to parity bits, cache tags, dirty bits, or SRAM configured as cache.

Memory Protection and Properties

This section describes the Memory Management Unit (MMU), memory pages, CPLB management, MMU management, and CPLB registers.

Memory Management Unit

The Blackfin processor contains a page based Memory Management Unit (MMU). This mechanism provides control over cacheability of memory ranges, as well as management of protection attributes at a page level. The MMU provides great flexibility in allocating memory and I/O resources between tasks, with complete control over access rights and cache behavior.

The MMU is implemented as two 16-entry Content Addressable Memory (CAM) blocks and two default descriptors. Each entry is referred to as a Cacheability Protection Lookaside Buffer (CPLB) descriptor. When enabled, every valid entry in the MMU is examined on any fetch, load, or store operation to determine whether there is a match between the address being requested and the page described by the CPLB entry. If a match occurs, the cacheability and protection attributes contained in the descriptor are used for the memory transaction with no additional cycles added to the execution of the instruction. If no valid CPLB entry matches, the cacheability and protection attributes are provided by the default descriptor.

Because the L1 memories are separated into instruction and data memories, the CPLB entries are also divided between instruction and data CPLBs. Sixteen CPLB entries and one default descriptor are used for instruction fetch requests; these are called *ICPLBs*. Another sixteen CPLB entries are used for data transactions; these are called *DCPLBs*. The ICPLBs and DCPLBs are enabled by setting the appropriate bits in the L1 Instruction Memory Control (`L1_IM_ICTL`) and L1 Data Memory Control (`L1_DM_DCTL`) registers, respectively.

Data accesses to system and core MMR space are never controlled by CPLBs. Accesses to all memory spaces including, when enabled, extended data accesses are controlled by the data CPLBs if the data CPLBs are enabled.

Instruction CPLB

The Instruction CPLB controls instruction fetches. Each of the 16 CPLB page descriptors consists of a pair of 32-bit values:

- `L1_IM_ICPLB_ADDR[n]` defines the start address of the page described by the CPLB descriptor. For more information, see Instruction Memory CPLB Address Registers.
- `L1_IM_ICPLB_DATA[n]` defines the properties of the page described by the CPLB descriptor. For more information, see Instruction Memory CPLB Data Registers.

The `L1_IM_ICPLB_DFLT` register provides default properties should no valid page descriptor match. For more information, see Instruction Memory CPLB Default Settings Register.

NOTE: To ensure proper behavior and future compatibility, all reserved bits in the L1_IM_ICPLB_DATAx register must be set to 0 whenever an L1_IM_ICPLB_DATAx register is written.

To ensure proper behavior and future compatibility, all reserved bits in the L1_IM_ICPLB_DFLT register must be set to 0 whenever an L1_IM_ICPLB_DFLT register is written.

Data CPLB

The Data CPLB controls data accesses by load and store instructions. Each of the 16 CPLB page descriptors consists of a pair of 32-bit values:

- The L1_DM_DCPLB_ADDR[m]defines the start address of the page described by the CPLB descriptor. For more information, see the Data Memory CPLB Address Registers.
- L1_DM_DCPLB_DATA[m] defines the properties of the page described by the CPLB descriptor. For more information, see the Data Memory CPLB Data Registers.

The L1_DM_DCPLB_DFLT register provides default properties should no valid page descriptor match. For more information, see the Data Memory CPLB Default Settings Register.

NOTE: To ensure proper behavior and future compatibility, all reserved bits in the L1_DM_DCPLB_DATAx register must be set to 0 whenever an L1_DM_DCPLB_DATAx register is written.

To ensure proper behavior and future compatibility, all reserved bits in the L1_DM_DCPLB_DFLT register must be set to 0 whenever an L1_DM_DCPLB_DFLT register is written.

Memory Pages

The 4G byte address space of the processor can be divided into smaller ranges of memory or I/O referred to as memory pages. Every address within a page shares the attributes defined for that page. The architecture supports 11 different page sizes:

- 1K byte
- 4K byte
- 16K byte
- 64K byte
- 256K byte
- 1M byte
- 4M byte
- 16M byte
- 64M byte

- 256M byte
- 1G byte

Different page sizes provide a flexible mechanism for matching the mapping of attributes to different kinds of memory and I/O.

Memory Page Attributes

Each page is defined by a two-word descriptor, consisting of an address descriptor word `L1_xM_xCPLB_ADDR[n]` and a properties descriptor word `L1_xM_xCPLB_DATA[n]`. The address descriptor word provides the base address of the page in memory. Pages must be aligned on page boundaries that are an integer multiple of their size. For example, a 4M byte page must start on an address divisible by 4M byte; whereas a 1K byte page can start on any 1K byte boundary. The second word in the descriptor specifies the other properties or attributes of the page. These properties include:

- Page size: any power of 4 between 1K byte and 1G byte.
- Cacheability properties
  - Cacheable/non-cacheable: Accesses to this page use the cache or bypass the cache.

    Cacheability may be specified separately for L1 and L2 cache. The L2 cache properties have no effect on derivatives without an L2 cache.
  - If cacheable: write-through/write-back.

    Data writes propagate directly to memory or are deferred until the cache line is reallocated.
  - If non-cacheable: I/O Device Space or regular memory.

    Data reads from I/O Device Space are non-speculative. This is suitable for use with memory mapped devices with read side-effects, but is considerably less efficient than a regular memory access so should not be used indiscriminately.
- Protection properties
  - Supervisor write access permission

    Enables or disables writes to this page when in Supervisor mode.

    Data pages only.
  - User write access permission

    Enables or disables writes to this page when in User mode.

    Data pages only.
  - User read access permission

    Enables or disables reads from this page when in User mode.

- Cache entry status
  - Valid

    The processor ignores the CPLB page descriptor unless this bit is set.
  - Dirty: The data in this page in memory has changed since the CPLB was last loaded.

    Writes to a page without this bit set cause a CPLB protection exception. Software is responsible for setting the bit to enable writes to the page and for propagating any subsequent modifications to the page further down the memory hierarchy. Ensure this bit is always set in data CPLBs if it is not required to track modifications to individual pages.
  - Lock

    Keep this entry in MMR; do not participate in CPLB replacement policy. This bit is ignored by the hardware and reserved for use by software implementing the CPLB replacement policy.

Default Memory Attributes

Cacheability and protection properties may be set for memory spaces not described by the CPLB page descriptors. After the processor has found no valid CPLB page descriptor matching an address, it looks up the default properties in `L1_IM_ICPLB_DFLT` register, for instruction fetches, or `L1_DM_DCPLB_DFLT` register, for data accesses.

The registers specify properties for L1 and non-L1 (system) memory separately.

- EOM: No Exception on Miss

  When this bit is cleared an access to the memory space (L1 or system) which does not match a valid CPLB page descriptor causes a CPLB miss exception. When this bit set the properties from this register are used when no valid CPLB page descriptor matches.
- Protection properties
  - Supervisor write access permission. Data CPLB only.
  - User write access permission. Data CPLB only.
  - User read access permission.
- Cacheability properties: system memory only
  - Cacheable/non-cacheable, in L1 and/or L2 cache
  - If cacheable: write-through/write-back. Data CPLB only.
  - If non-cacheable: I/O Device Space/Regular memory. Data CPLB only.

The default memory properties are only used when CPLBs are enabled.

CPLB Management

Use of the CPLB (cacheability protection lookaside buffer) is optional. If all L1 memory is configured as SRAM and no memory protection is required by the application, then CPLBs need not be enabled. However, CPLBs must be used if cache or memory protection is required.

NOTE: Before caches are enabled, the MMU and its supporting data structures must be set up and enabled.

Upon reset, the CPLBs are disabled, and the Memory Management Unit (MMU) is not used. CPLBs are enabled separately for instruction fetch and data accesses using:

- The Enable CPLB (`ENCPLB`) bit in the `L1_DM_DCTL` register to enable the data CPLB for data accesses, or
- The Enable CPLB (`ENCPLB`) bit in the `L1_IM_ICTL` register to enable the instruction CPLB for instruction fetches.

Once the CPLBs are enabled, an exception occurs when the Blackfin processor issues a memory operation for which no valid CPLB page descriptor exists in an MMR pair, and the default CPLB register indicates EOM (exception on miss). This exception places the processor into Supervisor mode and vectors to the MMU exception handler. The handler is typically part of the operating system (OS) kernel that implements the CPLB replacement policy.

The MMR storage locations for CPLB entries are limited to 16 page descriptors for instruction fetches and 16 page descriptors for data load and store operations.

For small and/or simple memory models, it may be possible to define a set of CPLB page descriptors combined with defaults that fit into these 32 entries, cover the entire addressable space, and never need to be replaced. This type of definition is referred to as a static memory management model.

However, operating environments commonly define more CPLB descriptors to cover the addressable memory and I/O spaces than will fit into the available on-chip CPLB MMRs. When this happens, a memory-based data structure, called a Page Descriptor Table, is used; in it can be stored all the potentially required CPLB descriptors. The specific format for the Page Descriptor Table is not defined as part of the Blackfin processor architecture. Different operating systems, which have different memory management models, can implement Page Descriptor Table structures that are consistent with the OS requirements. This allows adjustments to be made between the level of protection afforded versus the performance attributes of the memory-management support routines.

NOTE: Before CPLBs are enabled, valid CPLB page descriptors, or defaults, must be in place for both the Page Descriptor Table and the MMU exception handler. The `LOCK` bits of these CPLB page descriptors are commonly set so they are not inadvertently replaced in software.

The MMU exception handler uses the faulting address to index into the Page Descriptor Table structure to find the correct CPLB descriptor data to load into one of the on-chip CPLB page descriptor register pairs. If all on-chip registers contain valid CPLB entries, the handler selects one of the descriptors to be replaced, and the new descriptor information is loaded. Before loading new descriptor data into any CPLBs, the corresponding group of sixteen CPLBs must be disabled using the `ENCPLB` bit in `L1_DM_DCTL` or `L1_IM_ICTL`.

After the new CPLB page descriptor is loaded, the CPLB is re-enabled, the exception handler returns, and the faulting memory operation is restarted. This operation should now find a valid CPLB descriptor for the requested address, and it should proceed normally.

A single instruction may generate an instruction fetch as well as one or two data accesses. It is possible that more than one of these memory operations references data for which there is no valid CPLB page descriptor or default. In this case, the exceptions are prioritized and serviced in this order:

- Instruction page miss
- A page miss on DAG0
- A page miss on DAG1

CrossCore® Embedded Studio provides an MMU exception handler and automatic generation of the Page Descriptor Table structure. Please refer to *Cache and CPLBs* of the *System Run-Time Documentation*.

CPLB Status Registers

Bits in the DCPLB Status register (`L1_DM_DSTAT`) and ICPLB Status register (`L1_IM_ISTAT`) provide information about the cause of CPLB-related exceptions, and help identify the CPLB entry that has triggered the exception. The exception service routine may also inspect the `EXCAUSE` field of `SEQSTAT` and the CPLB entries to infer the cause of the fault. For more information, see the Data Memory CPLB Status Register and the Instruction Memory CPLB Status Register.

The `L1_DM_DSTAT` *and* `L1_IM_ISTAT` *registers are valid only while in the faulting exception service routine.*

DCPLB and ICPLB Fault Address Registers

The DCPLB Fault Address register (`L1_DM_DCPLB_FAULT_ADDR`) and ICPLB Fault Address register (`L1_IM_ICPLB_FAULT_ADDR`) hold the address that has caused a fault in the L1 Data Memory or L1 Instruction Memory, respectively. For more information, see the Data Memory CPLB Fault Address Register and the Instruction Memory CPLB Fault Address Register.

L1 Parity Protection

The following sections provide details of L1 parity error support on Blackfin processors.

Parity protection is an optional feature of the Blackfin architecture, processors with L1 parity protection have the `L1PARITY` bit set in the read-only `FEATURE0` register.

Parity Protection Coverage

Data and Instruction L1 SRAM are parity-protected using one bit per byte. Data and instruction L1 cache tag arrays and dirty bits are also parity-protected using at least one bit per byte. The details of parity coverage may vary between derivatives.

The parity bits are calculated and stored on every write to L1 SRAM or cache, either by the processor or by system traffic such as DMA and cache fill requests. Parity error checking is disabled by default. On powerup and reset L1 SRAM and the parity bits are in an undefined state. Software must write to all locations of L1 to initialize the memory and parity bits before enabling parity read checking.

Parity Error Detection and Notification

Parity error checking may be enabled separately for L1 Instruction memory and for L1 Data memory by setting the `RDCHK` (Read Parity Checking Enable) bit in the `L1_IM_ICTL` or `L1_DM_DCTL` registers respectively.

Parity errors are checked for whenever L1 memory is read. Parity checking is distributed so that all simultaneous L1 read traffic (DAG reads, Instruction reads, DMA reads, Victim reads, Flush reads, and MMR-initiated reads) can be simultaneously examined.

If a parity error is detected during any L1 read, an NMI is raised. The error is immediately signaled to the processor, even if the read is speculative in nature.

L1 reads by the processor are intercepted before they lead to further immediate consequence. The core will receive an NMI, be immediately stalled, and will remain stalled until it vectors to the handler in response to the NMI. This guarantees that core state is not modified based on corrupted L1 memory state.

If the read was not initiated by the local processor, an NMI is raised but the transfer is unstoppable. To guarantee that system is not corrupted by an L1 parity error, external reads of L1 by DMA or another processor should not be used, and write-back cache should also not be used. Note that write-through cache is a safe alternative to write-back cache, since it does not generate victim traffic.

The core will signal double fault error to the SEC if a parity error is detected while servicing a Reset or Emulation event. An NMI is not raised if a Parity Error is detected while servicing an NMI but the Parity Error Status registers are updated.

Parity Error Recovery

When a parity error is detected on an L1 read, the L1 memory has already been corrupted and must be restored to a known good state. Read-only data in SRAM might be restored from a known good copy in ROM or ECC protected L2 or L3 memory. Volatile data may need to be recomputed by rerunning a computation from a check pointed good state. Caches can be recovered by completely clearing the cache (by disabling and then immediately re-enabling it) so that all lines would be reacquired from non-L1 memory.

The location of a parity error can be determined by inspecting the SEQSTAT register and the L1_IM_IPERR_STAT (Instruction Memory Parity Error Status) or L1_DM_DPERR_STAT (Data Memory Parity Error Status) registers. For more information, see the Instruction Parity Error Status Register and the Data Memory Parity Error Status Register.

Four bits in the SEQSTAT register indicate whether the parity error occurred in instruction or data memory and whether it was detected on a read by the processor or a read by the system.

- PEIC indicates a parity error on a L1 Instruction Memory read by the processor.
- PEDC indicates a parity error on a L1 Data Memory read by the processor.
- PEIX indicates a parity error on a L1 Instruction Memory read by the system.
- PEDX indicates a parity error on a L1 Data Memory read by the system.

Once the general location of the error has been identified, the precise location can be discovered be reading L1_IM_IPERR_STAT for an error in L1 Instruction Memory, or L1_DM_DPERR_STAT for an error in L1 Data Memory.

If the fault is in SRAM the LOCATION field is set to zero, the ADDRESS field is set to bits 21 to 3 of the aligned address of the 8-bytes containing the faulting location, and a bit is set in the BYTELOC for each byte of the 8 bytes that are in error. The low numbered bit in BYTELOC is set if the low addressed byte is in error and so on.

When a misaligned memory read crosses an 8-byte boundary and errors occur in both parts then the only the errors in the low addressed part are reported.

If the fault is in a cache tag array, the LOCATION field is set to a non zero value. In this case, knowledge of the memory microarchitecture, which may vary between different Blackfin derivatives, is required to interpret the LOCATION, ADDRESS and BYTELOC fields. See Extended Data Access to L1 Caches.

The LOCATION and BYTELOC fields of the L1_IM_IPERR_STAT and L1_DM_DPERR_STAT registers are sticky, set when hardware encounters a parity error and only cleared on reset or when the processor explicitly writes to the MMR. The bits are "write-one-clear" and so can be cleared by writing back the value read from the registers.

If a parity error is detected while servicing an NMI or higher priority event the Parity Error Status registers are set. When the processor is at thread level or servicing a lower priority event, and bits in the LOCATION or BYTELOC fields are set then an NMI is raised. Thus it is recommended that on entering the NMI handler due to a parity error the Parity Error Status registers are read and written back to clear. If any further parity errors are detected while still in the handler they will set the status registers again, and as soon as the handler is exited it will be reentered to service the new error.

Parity Errors Simultaneous with Exceptions and Interrupts

The CPARINT (Concurrent Parity Error and Interrupt) bit, in SEQSTAT indicates a parity error was detected simultaneously with an exception or an interrupt.

An instruction that causes a parity error may also cause an exception. In this case CPARINT is set, IPEND[3] and IPEND[2] are set, RETN is set to the address of the exception handler and RETX to the instruction that cause the parity error. It may not be possible to recover from this situation as the exception may actually be a side effect of the parity error if it caused an instruction to be corrupted.

An interrupt may be latched at the same time as a parity error is detected. In this case CPARINT is set, IPEND[3] and the IPEND bit for the interrupt are set, RETN is set to the address of the interrupt handler and RETX to the instruction that caused the parity error. This enables the interrupt handler to be executed on return from the NMI handler, but also means RETN should not be inspected to determine the location of a parity error on an instruction read. L1_IM_IPERR_STAT should be used as always.

Direct Access To Parity Bits for L1 SRAM

In support of both parity error interrupt servicing and hardware test, it is possible to inspect the state of a parity bit without generating a parity error, and write the state of a parity bit to an intentionally incorrect state, by reading and writing to extended L1 memory locations.

When extended data access to L1 memory is enabled (ENX bit in the L1_DM_DCTL register), a number of extended accesses are allowed. These are listed in the **Permitted Extended Accesses** table.

Table 7-1: Permitted Extended Accesses

Access Type	Address	Parity
Read	L1 SRAM address + 0x80000	Read data at address without parity checking
Read	L1 SRAM address + 0xC0000	Read parity bits for 32-bit word at address
Write	L1 SRAM address + 0xC0000	Invert parity bits for 32-bit word at address to create an intentional parity error for test purposes

Parity bits are read into the low order bit of each byte. The value of remaining bits may not be zero and cannot be relied upon.

Direct access to parity bits for L1 caches tags and dirty bits is possible when extended data access is enabled, and not possible otherwise. Knowledge of the memory microarchitecture, which may vary between different Blackfin derivatives, is required to do this. See Extended Data Access to L1 Caches.

L1 Initialization Requirements

If parity checking is to be used, software must write all locations of L1 after each processor power-up. This includes initial device power-up as well as subsequent exits from Hibernate. Normally, this process takes place in the processor's boot code. These writes are necessary to initialize the otherwise random states of parity bits to legitimate states. All of L1 must be initialized, rather than just those locations expected to be read, otherwise speculative accesses have the potential to trigger unintended parity errors.

To allow the processor to be able to initialize L1 without risk of triggering these unwanted parity errors, L1 parity error checking on reads is disabled by default. Writes are always performed with parity bits calcu-

lated and stored. This mode is controlled by the RDCHK (Read Parity Checking Enable) bit in the L1_IM_ICTL and L1_DM_DCTL registers. This bit is deasserted by hardware reset and must be set by software (after L1 initialization) to enable read parity checking. Initialization of L1 may be achieved through any combinations of DMA or processor L1 accesses.

When the cache is enabled all the tags are written to, so enabling the cache also initializes the parity bits protecting the tags. It does not initialize the parity bits for the SRAM holding the cache data arrays. These must be initialized prior to enabling the cache as part of the L1 SRAM initialization process described above.

Additional Notes on Parity Errors

Although hardware provides a means for determining the locations of corrupted L1, this may not provide sufficient clues to identify the locations outside the core that receive corrupted data from L1. The system can receive corrupted L1 data either through direct access by another core or DMA or cache victimization. In the case of direct system access, an L1 source address may not alone implicate the system target address. In the case of cache victimization, the victim address stored in the Tag array is typically overwritten with a Fill address prior to all victim data being extracted from L1 memory. Therefore, cache will not natively be able to provide the system address of a corrupt victim.

A cache bypass mode is provided to allow direct access to system memory. Cache bypass is enabled by setting the CBYPASS bit in the L1_IM_ICTL register to bypass instruction cache, and in the L1_DM_DCTL to bypass data cache. This can be useful during servicing of L1 parity errors. While cache is bypassed, CPLB cache settings are ignored, and cache hits are blocked. This preserves the cache in the precise state it was in prior to being bypassed, ignoring any already active fill, flush and victimization servicing, which continue until complete. While cache is being bypassed, extended data accesses may still be used to access cache content.

Example Parity Handler

```
.section NONCACHED_ECC_PROTECTED_DATA;
.var saved_sp;
.var nmi_handler_stack[BIG_ENOUGH];

.section NONCACHED_ECC_PROTECTED_CODE;
.extern nmi_handler;
nmi_handler:
/* switch to ECC protected stack */
[saved_sp] = SP;
SP = nmi_handler_stack + BIG_ENOUGH;
/* save other registers on ECC protected stack */

/* If program used system MMRs or memory with read side effects, check for non-
speculative access abort. */
R7 = SEQSTAT;
CC = BITTST(R7, NSPECABT);
```

```
IF CC JUMP unrecoverable_error;

/* If there are other sources of NMI, check for external NMI which vectors to the same
handler as parity. */
CC = BITTST(R7, SYSNMI);
IF CC JUMP nmi_handler;

/* CPARINT indicates parity error simultaneous with exception or interrupt, not
necessarily recoverable. */
CC = BITTST(R7, CPARINT);
IF CC JUMP unrecoverable_error;

/* If DMA may have read L1 or Write-back cache is enabled, check for parity error on
system read */
CC = BITTST(R7, PEIX);
IF CC JUMP unrecoverable_error;
CC = BITTST(R7, PEDX);
IF CC JUMP unrecoverable_error;

/* We have a recoverable parity error. */
CC = BITTST(R7, PEIC);
IF CC JUMP parity_in_instruction_L1;

/* Parity error in data L1. */
R7 = [L1_DM_DPERR_STAT];
/* clear the error by writing BYTELOC and LOCATION */
[L1_DM_DPERR_STAT] = R7;
/* test for error in cache TAG or MOD */
R5 = R7 << 29;
CC = R5 == 0;
IF !CC JUMP parity_error_in_data_cache;

/* compute the error address */
R6 = [L1_DM_SRAM_BASE_ADDRESS];
R5 = R7 << 8;
R5 = R5 >> 8;
R6 = R6 + R5;
/* if address is in non-cache sram goto reload */

/* otherwise reset data cache */
parity_in_data_cache:
R7 = [L1_DM_DCTL];
R6 = R7;
BITCLR(R6, 3);
BITCLR(R6, 2);
[L1_DM_DCTL] = R6; /* disable cache */
CSYNC;
[L1_DM_DCTL] = R7; /* reenable cache */
CSYNC; /* wait for cache to reinitialise */
JUMP return_from_parity_error;

parity_in_instruction_L1:
R7 = [L1_IM_IPERR_STAT];
```

```
/* clear the error by writing BYTELOC and LOCATION */
[L1_IM_IPERR_STAT] = R7;
/* test for error in cache TAG or MOD */
R5 = R7 << 29;
CC = R5 == 0;
IF !CC JUMP parity_error_in_instruction_cache;
/* compute the error address */
R6 = [L1_DM_SRAM_BASE_ADDRESS];
R5 = R7 << 8;
R5 = R5 >> 8;
R6 = R6 + R5;
/* if address is in non-cache sram goto reload */

/* otherwise reset instruction cache */
parity_in_instruction_cache:
R7 = [L1_IM_ICTL];
BITCLR(R7, 1);
[L1_IM_ICTL] = R7; /* disable cache */
CSYNC;
BITSET(R7, 1);
[L1_IM_ICTL] = R7; /* reenable cache */
CSYNC; /* wait for cache to reinitialise */

return_from_parity_error:
/* restore registers */
SP = [saved_sp];  /* restore stack */
RTN;
```

Memory Transaction Model

Both internal and external memory locations are accessed in little endian byte order. The Data Stored in Little Endian Order figure shows a data word stored in register R0 and in memory at address location *addr*. B0 refers to the least significant byte of the 32-bit word.

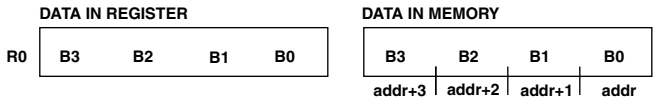


Figure 7-5: Data Stored in Little Endian Order

The Instructions Stored in Little Endian Order figure shows 16- and 32-bit instructions stored in memory. The diagram on the left shows 16-bit instructions stored in memory with the most significant byte of the instruction stored in the high address (byte B1 in *addr+1*) and the least significant byte in the low address (byte B0 in *addr*).

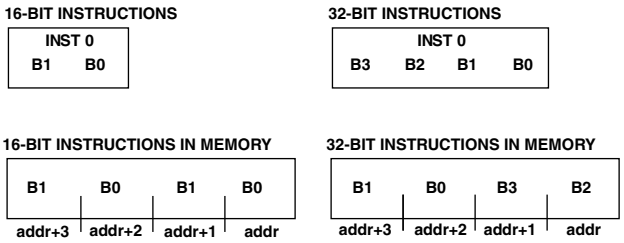


Figure 7-6: Instructions Stored in Little Endian Order

The diagram on the right shows 32-bit instructions stored in memory. Note the most significant 16-bit half word of the instruction (bytes B3 and B2) is stored in the low addresses (*addr+1* and *addr*), and the least significant half word (bytes B1 and B0) is stored in the high addresses (*addr+3* and *addr+2*).

Load/Store Operation

The Blackfin processor architecture supports the RISC concept of a Load/Store machine. This machine is the characteristic in RISC architectures whereby memory operations (loads and stores) are intentionally separated from the arithmetic functions that use the targets of the memory operations. The separation is made because memory operations, particularly instructions that access off-chip memory or I/O devices, often take multiple cycles to complete and would normally halt the processor, preventing an instruction execution rate of one instruction per cycle.

In write operations, the store instruction is considered complete as soon as it executes, even though many cycles may execute before the data is actually written to an external memory or I/O location. This arrangement allows the processor to execute one instruction per clock cycle, and it implies that the synchronization between when writes complete and when subsequent instructions execute is not guaranteed. Moreover, this synchronization is considered unimportant in the context of most memory operations.

Interlocked Pipeline

In the execution of instructions, the Blackfin processor architecture implements an interlocked pipeline. When a load instruction executes, the target register of the read operation is marked as busy until the value is returned from the memory system. If a subsequent instruction tries to access this register before the new value is present, the pipeline will stall until the memory operation completes. This stall guarantees that instructions that require the use of data resulting from the load do not use the previous or invalid data in the register, even though instructions are allowed to start execution before the memory read completes.

This mechanism allows the execution of independent instructions between the load and the instructions that use the read target without requiring the programmer or compiler to know how many cycles are actually needed for the memory-read operation to complete. If the instruction immediately following the load uses the same register, it simply stalls until the value is returned. Consequently, it operates as the programmer expects. However, if four other instructions are placed after the load but before the instruc-

tion that uses the same register, all of them execute, and the overall throughput of the processor is improved.

Alignment

Nonaligned memory operations are supported. Loads and stores with addresses which are not a multiple of the data size, access the bytes at sequential addresses starting with the address passed to the instruction, as expected. This may generate multiple memory read or write operations, but generally the instruction will not take more cycles than the equivalent two aligned loads and stores.

Aligned addresses are required in special circumstances, such as access to MMRs, I/O Device space, Exclusive loads and stores and Extended data access. In these cases an address which is not a multiple of the data size cause a Misaligned Address exception.

For backward compatibility, some instructions in the quad 8-bit group and those used with the `DISALGNEXCPT` instruction do not cause alignment exceptions, but ignore the low order bits of a nonaligned address to access aligned data.

Ordering of Loads and Stores

The relaxation of synchronization between memory access instructions and their surrounding instructions is referred to as weak ordering of loads and stores. Weak ordering implies that the timing of the actual completion of the memory operations-even the order in which these events occur-may not align with how they appear in the sequence of the program source code. All that is guaranteed is:

- Load operations will complete before the returned data is used by a subsequent instruction.
- Load operations using data previously written will use the updated values.
- Store operations will eventually propagate to their ultimate destination.

Because of weak ordering, the memory system is allowed to prioritize reads over writes. In this case, a write that is queued anywhere in the pipeline, but not completed, may be deferred by a subsequent read operation, and the read is allowed to be completed before the write. Reads are prioritized over writes because the read operation has a dependent operation waiting on its completion, whereas the processor considers the write operation complete, and the write does not stall the pipeline if it takes more cycles to propagate the value out to memory. This behavior could cause a read that occurs in the program source code after a write in the program flow to actually return its value before the write has been completed. This ordering provides significant performance advantages in the operation of most memory instructions.

Speculative Load Execution

Load operations from memory do not change the state of the memory value. Consequently, issuing a speculative memory-read operation for a subsequent load instruction usually has no undesirable side effect. In some code sequences, such as a conditional branch instruction followed by a load, performance

may be improved by speculatively issuing the read request to the memory system before the conditional branch is resolved. For example,

```
  IF CC JUMP away_from_here
  R0 = [P2];
  . . .
away_from_here:
```

If the branch is taken, then the load is flushed from the pipeline, and any results that are in the process of being returned can be ignored. Conversely, if the branch is not taken, the memory will have returned the correct value earlier than if the operation were stalled until the branch condition was resolved.

Store operations never access memory speculatively, because this could cause modification of a memory value before it is determined whether the instruction should have executed.

Interruptible Load Behavior

A load instruction may generate more than one memory-read operation, because it is interruptible. If an interrupt of sufficient priority occurs between the load instruction entering the pipeline and the completion of the load instruction, the sequencer cancels the instruction. After execution of the interrupt, the interrupted load is executed again. This approach minimizes interrupt latency. However, it is possible that a memory-read operation was initiated before the load was canceled, and this would be followed by a second read operation after the load is executed again. For most memory accesses, multiple reads of the same memory address have no side effects.

There is no corresponding issue with store instructions, as the memory-write operation only happens after a store instruction has committed. So the store can only be interrupted before the memory-write has been initiated, and is canceled with no visible side effect. The store instruction is re-executed on return from the interrupt and ultimately initiates the memory-write.

Hazards of the High Performance Memory Architecture

The Blackfin memory model, with weak ordering of reads and writes and redundant read operations, enables long pipeline while avoiding stalls and maintaining fast interrupt response. However, it can cause side effects that the programmer must be aware of to avoid improper system operation.

When sharing data with another core or device via memory accessible to both, the order of how read and write operations complete is often significant, as the writing core must be sure the data is visible to the reading device before signaling that the data is available.

Similarly, when writing to or reading from non memory locations such as off-chip I/O device registers, the order of how read and write operations complete is also significant. For example, a read of a status register may depend on a write to a control register. If the address is the same, the read would return a value from the store buffer rather than from the actual I/O device register, and the order of the read and write at the register may be reversed. Both these effects could cause undesirable side effects in the intended operation of the program and peripheral.

Redundant memory-reads can also be an issue. Interruptible load behavior can cause many memory-read operations where one was intended. For most memory accesses, multiple reads of the same memory address have no side effects. However, for some off-chip memory-mapped devices, such as peripheral data FIFOs, reads are destructive. Each time the device is read, the FIFO advances, and the data cannot be recovered and re-read. The redundant memory-reads due to speculation will also cause problems if the load from a peripheral with destructive read behavior, such as a FIFO, is subsequently aborted.

Speculation can also be a problem where a load from an illegal address is aborted. A redundant memory-read operation from a non-L1 address which does not map to any memory or device in the system will cause an External Memory Addressing Error. Note the load might be aborted because it is in the shadow of an conditional jump that tests whether the address is valid.

In summary following hazards exist.

- Reordering of externally visible write with another externally visible action.
- Reordering of externally visible read and write to same location.
- Reordering of externally visible read and write to different locations.
- Caches and write buffers can prevent memory operations becoming externally visible at all.
- Destructive reads not generated because serviced from the write buffer, or cache.
- Repeated destructive reads due to interruptible loads.
- Unintended destructive reads due to speculative loads.
- Unintended access to illegal addresses causing spurious error interrupts.

The Blackfin architecture provides a number of solutions to these problems. Synchronization instructions (`CSYNC` or `SSYNC`) may be used to impose a precise ordering at the points in the code where it is required while retaining the benefits of weak ordering generally.

Cacheability properties may be specified in the CPLBs, which control the external visibility of memory operations and specify some regions are I/O Device space. Loads from MMRs and I/O Device space are never executed speculatively and are non-interruptible. All reads and writes to MMRs are strongly ordered.

CPLBs may be used to avoid spurious illegal address exceptions, as the memory-read operation will not be initiated for a load that causes a page miss exception, but the exception will be suppressed in the case of a speculative load.

Synchronizing Instructions

When strong ordering of loads and stores is required, as may be the case for sequential accesses to shared memory, use the core or system synchronization instructions, `CSYNC` or `SSYNC`, respectively.

The CSYNC instruction ensures all pending core operations have completed and the store buffer (between the processor core and the L1 memories) has been flushed before proceeding to the next instruction. Pending core operations may include any pending interrupts, speculative states (such as branch predictions), or exceptions.

Consider the following example code sequence:

```
  IF CC JUMP away_from_here;
  CSYNC;
  R0 = [P0];
away_from_here:
```

In the preceding example code, the CSYNC instruction ensures:

- The conditional branch (IF CC JUMP away_from_here) is resolved, forcing stalls into the execution pipeline until the condition is resolved and any entries in the processor store buffer have been flushed.
- All pending interrupts or exceptions have been processed before CSYNC completes.
- The load is not fetched from memory speculatively.

The SSYNC instruction ensures that all side effects of previous operations are propagated out through the interface between the L1 memories and the rest of the chip. In addition to performing the core synchronization functions of CSYNC, the SSYNC instruction flushes any write buffers between the L1 memory and the system domain and generates a sync request to the system that requires acknowledgment before SSYNC completes.

Where the external visibility of memory operations or interaction with system MMRs is a concern, SSYNC should be used. CSYNC is sufficient to control interaction with core MMRs and the L1 memory system.

Cache Coherency

For shared data, software must provide cache coherency support as required. To accomplish this, use the FLUSH instruction (see the FLUSH description in Data Cache Control Instructions), and/or explicit line invalidation (see Data Cache Invalidation).

Whenever the external visibility of reads and writes is a concern, the cacheability properties specified in the CPLBs must be considered. A store to write-back L1 cache is only guaranteed to become visible externally after a FLUSH instruction is executed, whereas stores to write-through L1 is visible after the memory-write completes.

If the memory region is written by an another core or device, there is no automatic coherence mechanism to ensure an earlier values in the cache are invalidated. So a load operation might return the stale data from the cache. Explicit line invalidation could be used to ensure coherency.

Commonly shared data which is updated by more than one writer is maintained in non-cacheable regions.

I/O Device Space

The I/O Device space property may be specified in the CPLBs. Read operations from memory with this property are modified in a manner more suitable for access to devices with destructive reads, such as FIFOs.

I/O device space is not cached. Reads from I/O Device space are executed in a non-interruptible manner, and are never executed speculatively.

However writes to I/O Device space may be buffered. Reads will not be serviced from the write buffer, so a write followed by a read to the same location will always become externally visible and in the correct order. But reads and writes to different locations may get reordered unless they are separated by a `SSYNC` instruction.

Loads and stores to I/O Device space may not be executed in parallel with another memory operation, and addresses must be aligned to the data size.

Memory-Mapped Registers

A portion of the address space is reserved for MMRs (Memory Mapped Registers). This is split into a region for system MMRs and a region for core MMRs. Typical locations of these regions are system MMRs: `0x2000 0000-0x2FFF FFFF`, and core MMRs: `0x1FC0 0000-0x1FFF FFFF`. Some derivatives may differ. Refer to the specific *Blackfin+ Processor Hardware Reference* for your derivative.

All MMRs are accessible only in supervisor mode. Access to MMRs in user mode generates an Illegal use of supervisor resource exception. A load or store to an MMR may not be issued in parallel with another load or store, an attempt to do so also generates an Illegal use of supervisor resource exception. Loads from MMRs are non-speculative, and non-interruptible. All loads and stores to each MMR space are strongly ordered.

The core MMRs space is located at the same address on every Blackfin core in a system. Core MMRs may only be accessed by Load and Store instructions executed on the local core, and each core accesses its own core MMRs. Core MMRs cannot be accessed by DMA. Like non-memory mapped registers, the core MMRs connect to the 32-bit wide Register Access Bus (RAB). They operate at CCLK frequency.

All core MMRs must be read and written with aligned 32-bit accesses. However, some MMRs have fewer than 32 bits defined. In this case, the unused bits are reserved, and should be written as zero.

System MMRs connect through the system crossbars (SCBs). Accesses to system MMRs must be aligned on the data size.

Accesses to nonexistent MMRs generate an illegal access exception. The system ignores writes to read-only MMRs.

Each chapter describing a portion of the processor architecture includes a description of the related core MMRs.

Non-speculative, Non-interruptible, Reads

Loads from MMRs and I/O Device space are non-speculative and non-interruptible.

When a load from one of these spaces is encountered, a non-speculative request is sent to the sequencer. The sequencer will then disable interrupts and flush the pipeline below the load instruction. Once this flushing has completed, the read request is initiated. The non-speculative read is completed normally, and interrupts re-enabled, when the read data is returned.

However there are four interrupt levels (IVG0-IVG3) that are effectively non-maskable and therefore might interrupt a non-speculative read.

- Emulator hardware interrupt (IVG0): Emulator hardware interrupts can interrupt I/O accesses. However, this would only be in debugging situations.
- RESET (IVG1): Reset can interrupt non-speculative accesses, but since the core is being reset, this is expected behavior.
- NMI (IVG2): NMI's come from three possible sources:
  - External events
  - `RAISE 2` instruction (which is committed before I/O starts, so it cannot interrupt an non-speculative access)
  - Parity errors (which can interrupt an non-speculative access if caused by DMA of cache victim traffic)
- Exception (IVG3): Exceptions are always instruction related. The non-speculative read mechanism is designed to allow all exceptions in the pipeline before the non-speculative read to be taken before the non-speculative read is placed on the bus. So there will never be an exception during an non-speculative read.

If a non-speculative read is interrupted, whether to an I/O Device page or MMR, the `NSPECABRT` (Non-speculative access was aborted) bit is set in the `SEQSTAT` register. As the effect of the interrupted non-speculative read might be that a read side effect occurred but the read data was lost, this is a non-recoverable error condition.

Exclusive Load, Store, and Sync (Spin Lock Example)

The load from memory exclusive, store to memory exclusive, and synchronize exclusive state instructions permit implementing software semaphores to control the interaction between tasks on separate processor cores or to separate tasks running on a single processor core.

A load exclusive instruction reads data from memory in the same manner as a regular load instruction. The load exclusive also establishes exclusive access to that memory location. A store exclusive instructions only modifies memory if the task still has exclusive access to the location. An intervening load exclusive from another task causes the exclusive access to be lost. The state of exclusive operations is tracked in the

XMONITOR, XWACTIVE and XWAVAIL bits in the SEQSTAT register. The SYNCEXCL instruction synchronizes this state with the processor state, ensuring the CC bit in ASTAT has been updated to indicate whether a previous store exclusive instruction was performed successfully. The address passed to an exclusive load or store must be aligned to the size of the data.

The following code sequence implements a spin lock:

```
P0 = lock;   /* address of lock */
R1 = 1;      /* lock value */
spin:
R0 = B[P0] (Z,EXCL);
CC = R0==0;                  /* is semaphore unlocked? */
if !CC JUMP spin;            /* no - try again */
CC = (B[P0] = R1) (EXCL);    /* try to lock */
SYNCEXCL;   /* wait for write and copy to CC */
IF !CC JUMP spin;   /* failed - try again */

/* critical section */

R1 = 0;      /* unlocked value */
B[P0] = R1;  /* unlock */
```

Semaphores controlling interaction between tasks on separate cores should be placed in non-cacheable, non-L1 memory accessible by both cores. In this case the load and store exclusive instructions generate exclusive transactions on the system bus, and the memory controller participates in a protocol that ensures a store exclusive will fail if the memory location has been modified by another core since the corresponding load exclusive.

Semaphores controlling interaction between tasks on the same core may be placed in cacheable memory or L1 SRAM. In this case the load and store exclusive instructions do not generate special memory transactions. The SYNCEXCL instruction must be called in context switch code to clear any pending exclusive transactions and to preserve the result of any store exclusive in the CC bit of the preserved ASTAT register.

```
/* context switch */
SYNCEXL;
[--SP] = ASTAT; /* saves store excl result if one was pending */
```

Interrupt handlers that are known not to use exclusive operations may leave the exclusive state unmodified. Any pending exclusive write operations will complete and update the state in SEQSTAT which will be read by a SYNCEXCL instruction on return from the interrupt.

SYNCEXCL should also be used in the exception handler to reset exclusive state on exceptions caused by load or store exclusive instructions.

Load exclusive and store exclusive instructions must be aligned and may not be used with MMRs, I/O Device space, or extended data access.

Execution results for exclusive load instructions and exclusive store instructions vary, depending on whether the memory addressed is shareable or non-shareable. The share-ability of memory spaces is determined from the memory space and the CPLB settings as shown in the **Memory Kinds** table.

An exclusive load or exclusive store to an *illegal* memory location causes an exception. An exclusive load or exclusive store to a *non-shareable* memory location succeeds, but the operation is not exclusive with respect to other cores. The operation is exclusive with respect to other threads running on the core executing the instruction. An exclusive load or exclusive store to a *shareable* memory location ensures exclusivity wit respect to other cores by using exclusive transactions on the memory bus. Exclusive transactions require hardware support in the memory device and if the support is not available an uncached exclusive load from that memory will cause an exception.

Table 7-2: Memory Kinds (Example for ADSP-BF70x Processors)

Memory	CPROPS	Meaning	Share-ability
MMR	any	Core or system MMR	Illegal
L1	any	L1 SRAM	Non-shareable
non-L1	CPLBEN=0	CPLB Disabled	Shareable
non-L1	CPLBBYPASS=1	Cache temporarily disabled	Shareable
non-L1	000	Page is non-cacheable memory	Shareable
non-L1	001	Non-cacheable in L2, Write Back Cacheable in L1	Non-shareable
non-L1	010	Write Back Cacheable in L2, Non-cacheable in L1	Non-shareable
non-L1	011	Write Back Cacheable in L1 and L2	Non-shareable
non-L1	100	I/O Device Space	Illegal
non-L1	101	Non-cacheable in L2, Write Through Cacheable in L1	Non-shareable
non-L1	110	Write Through Cacheable in L2, Non-cacheable in L1	Non-shareable
non-L1	111	Write Through Cacheable in L1 and L2	Non-shareable

Atomic TESTSET Instruction (Spin Lock Example)

The processor provides an atomic `TESTSET` instruction. This is primarily provided for backward comparability and it is recommended to use Exclusive Load and Store instructions which make more efficient use of system resources (if that is possible). The `TESTSET` instruction reads an indirectly addressed memory byte, tests whether it is zero, and then writes the byte back to memory with the most significant bit (MSB) set, all as one indivisible operation. If the byte is originally zero, the instruction sets the `CC` bit. If the byte is originally nonzero, the instruction clears the `CC` bit.

The `TESTSET` instruction is used with a regular store to implement a spin lock as follows.

```
P0 = lock;
spin:
TESTSET (P0);
IF !CC JUMP spin;

/* critical section */
```

```
R1 = 0;     /* unlock value */
B[P0] = R1; /* unlock */
```

NOTE: For more information, see the `TESTSET` instruction's reference page.

L1 Memory Microarchitecture

This section provides an overview of the L1 memory system in recent Blackfin processors. The details of L1 memory described in this section may vary between different derivatives, and should not be relied on in portable software.

L1 Memory Access

Processor access to the L1 memory space is intended to occur in a single cycle. The L1 memory has four virtual ports: DAG0 read, DAG1 read, Store, and DMA read/write. The memories used to implement L1 are physically single-ported, so the possibility exists for access conflicts. To reduce the likelihood of memory conflicts, the L1 memory is divided into individually accessible 4k subbanks. Each subbank has its own port multiplexer, and each port has a dedicated data bus. A memory conflict will only occur when multiple ports request at least one byte from the same subbank in the same cycle.

The incoming addresses for the four L1 ports are centrally decoded into subbank selects. The subbank selects are then compared for collisions. The collisions are prioritized, and the winner is allowed to access memory. The losers receive a stall, and try again in the next cycle.

Load instructions present the read address to the memory system in pipeline stage F (DF1). The memory read occurs in stage G (DF2). The result is returned to the processor History buffer in stage H (EX1). Store instructions are described in L1 Data Stores.

The CPLB data values are used to generate exceptions in pipe stage G (DF2). The exception information is pipelined along with the memory operation and deposited into the history buffer in pipe stage H (EX1).

DMA accesses use the same timing as DAG reads and writes, that is 3 cycles for reads and two for writes.

Memory Logical Subbank Arrangement

The L1 memory sub banks are logically arranged into rows and columns as shown in the **ADSP-BF70x Data Bank A Address Mapping to Sub Banks** figure. The data path width consists of the number of memory columns times the width of a sub bank. All sub banks are identical. The subbanks are four bytes wide, and there are two columns. Thus the data path width is eight bytes. The number of rows is determined by the desired amount of L1 memory.

The address provided by the DAGs is a byte address. The lower three bits of the address index bytes across the eight byte data path width. The next higher set of bits address memory words within a sub bank. In Data Blocks A and B, each sub bank is also logically split into an upper and lower half, with bit 14 selecting the half. Thus, address bits 31 to 20 select L1 memory and the block within L1 memory. Address bits 119

to 15 and 13, 12 and 2 select the sub bank. Address bit 14 and bits 11 to 3 select the row within the sub bank. Finally address bits 1 and 0 select the byte within the row. Data bank C only contains two sub banks, bit 2 selects the sub bank, and bits 13 to 3 select the row. So, the addresses are contiguous.

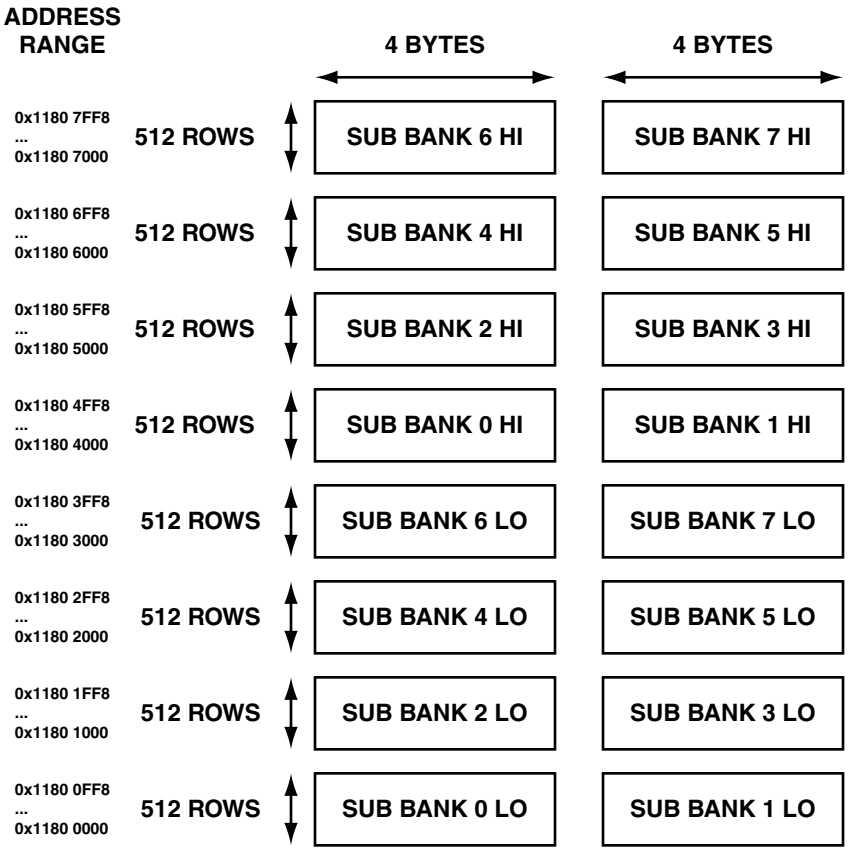


Figure 7-7: ADSP-BF70x Data Bank A Address Mapping to Sub Banks

Misaligned Data Access to L1

Misaligned memory access are supported. If all the bytes of an access lie within the address offset span 0.. 7, then the misaligned access can be serviced in a single cycle (assuming no subbank conflicts). If this requirement is not satisfied, then the access is broken into two pieces within the MMU. This is known as a "crossed" access. The two halves of a crossed access are handled sequentially. The halves of a crossed access are reassembled in the history buffer and returned to the core as a single entity. Thus the core is isolated from the effects of the misaligned access, aside from the extra cycle required to retrieve all the data.

L1 Data Stores

Processor stores to data memory are more complicated than loads because the store address arrives in pipe stage F (DF1), but the data does not arrive until pipe stage J (WB). The Memory Controller must create a placeholder for the address. When the data arrives, it is matched with the address. The address/data pair

is the delivered to L1 memory on the Store port. The block that records the write address is called the Read/Write Buffer.

The name Read/Write Buffer is derived from the fact that it handles uncached loads from the system as well as all stores. Each Read/Write buffer entry contains an address which describes a particular set of eight aligned memory bytes, and data storage for those bytes. The number of bytes stored in a given buffer is determined by the data path width. When a new store enters pipe stage G (DF2), its address is compared to the addresses in all currently valid Read/Write entries. If there is a match, then the existing buffer entry is used, otherwise a new buffer entry is allocated. The ID of the buffer entry is placed in a queue called the store queue. The store queue performs the function of matching incoming store data with the proper Read/Write entry. When valid write data is present in the Read/Write buffer, it gets scheduled for "draining" to L1 memory using the L1 Store port.

Write Gathering is supported. Consider a sequence of byte writes to L1 memory, starting with an aligned address. The first byte will allocate an Read/Write entry. The address of the second byte will match the Read/Write entry of the first. These two bytes will use the same Read/Write entry. This is called write gathering. The Read/Write buffer entry tracks which of the eight bytes has been loaded with valid data. Only these bytes are written to memory.

Write Data Forwarding is supported. Consider a byte write, followed by a read of the same byte. The read operation must wait for the write data to become available in the memory. Once the write data arrives at the memory system, it often takes several cycles to get the data into the L1 memory subbank. Write forwarding short circuits this process by delivering the write data to the history buffer as soon as it arrives at the memory controller.

System Memory Access

Unlike L1 loads, System loads allocate an Read/Write buffer entry. This is because, like stores, the data arrives late and we need an address placeholder and a way to associate the address and data. Unlike stores, each System load is allocated a new Read/Write entry. There is no "read gathering" allowed. The ID of the Read/Write entry allocated for the system read is passed to the System Read Queue and the history buffer. System reads gather in System Read Queue waiting to be dispatched to the System. When the System gasket accepts the read request, the request is popped off the queue. When the read data returns, still tagged with the Read/Write ID, it is forwarded directly to the history buffer.

System stores behave in a similar fashion to L1 stores. Multiple System stores into the same eight-byte aligned space will gather into the same Read/Write buffer entry. Write drains to the System gasket are scheduled the same way as L1 writes. When a System store causes an Read/Write entry to be allocated, subsequent System loads matching that Read/Write address will use that entry's ID. This is necessary to ensure that the write data is properly forwarded to the history buffer along with the bytes read from system memory.

System accesses may also be misaligned and result in a crossed access. In this case, two Read/Write entries will be allocated. The two components of the crossed access are treated as separate transactions by the Read/Write buffer and System gasket. The history buffer understands that crossed System reads are split, and rejoins them before delivery to the core.

Reads of MMRs and I/O Device space and the read part of the TESTSET instruction are non-speculative and non-interruptible. These reads are not added to the System Read Queue. Instead, when the read operation is passed to the history buffer in pipe stage H (EX1), an non-speculative request is sent to the sequencer. The sequencer will then disable interrupts and flush the pipeline below the non-speculative read, while holding the non-speculative read in stage H (EX1). Once this flushing has completed, the system read request is sent to the System Memory Interface. The non-speculative read is completed normally, and interrupts re-enabled, when the read data is returned.

Core MMR Access

Reads of MMR registers in the MMU use the same timing as L1 memory reads. MMR reads must be 32-bit aligned accesses, else an MMR exception is generated. MMR reads are stalled in pipe stage G (DF2) until all prior MMR writes have committed. This behavior is necessary because of the functional side effects of MMR writes. MMR writes take effect in the cycle immediately after write commit in J (WB). There is no MMR store queue which could create timing uncertainty. CPLBs do not control the MMR address space. Read/write access is always permitted in supervisor mode and denied in user mode.

Core MMR reads are non-speculative.

L1 Cache Details

The upper 16K bytes of L1 instruction SRAM, L1 Data Block A, and L1 Data Block B may be individually configured as cache. When so configured, these 16K bytes are reserved for cache lines fetched from non-L1 memory. The cache tags occupy bespoke memory only used when cache is enabled.

Both Data Block A and Data Block B can be configured as data cache. Address bit 14 or 23 of a cacheable load and or store selects which data cache is searched for a particular cache line, when both Block A and B are configured as cache. See Example of Mapping Cacheable Address Space.

When accessing a cache line present in L1 cache, the access timing is identical to a standard L1 memory access. On a cache read all lines in the set are read in parallel with the cache tag access and then the correct data is selected. On an instruction fetch all 8 subbanks are read, and a data load may access up to 4 subbanks. These accesses will collide with accesses to the other half of these subbanks, which remain in use as L1 SRAM.

Extended Data Access to L1 Caches

When extended data access is enabled (ENX bit in L1DM_DCTL) cache tags, dirty bits and associated parity bits may be directly accessed with load and store instructions.

The data cache tag memory and dirty array reside in the upper portion of the 256KB region of L1 Data Block A.

All loads and stores to these regions must be 32-bits wide, have an 8-byte aligned address, be issued from DAG0 (must not be in parallel with another load or store).

Two bits of the extended data access address encode PARCTL and PARSEL. Setting PARCTL disables parity checking for the access. When accessing L1 SRAM, setting PARSEL with PARCTL enables parity bits to be toggled, to cause parity errors for test purposes, when writing to the address, and enables SRAM parity bits to be read when reading from the address. When accessing Cache tag and dirty arrays, PARSEL has no meaning, as the parity bits can be directly read or written.

The following tables provide details of the data cache tags. Separate copies of each tag are maintained for accesses by DAG0 and for accesses by DAG1.

Table 7-3: Data Cache Tags extended data access address

Address Bit	Meaning
A[31:20]	L1 Data Block A address
A[19]	PARCTL
A[18]	PARSEL
A[17:14]	1111
A[13]	1=Tag for DAG0, 0=Tag for DAG1
A[12:4]	Set Index
A[3]	Way
A[2:0]	000

Table 7-4: Data Cache Tags extended data access value

Data Bit	Value
D[31:29]	0
D[28]	Priority Parity
D[27]	Fill Pending Parity
D[26]	Next Victim Parity
D[25]	Valid Parity
D[24]	Tag Parity
D[23]	Priority
D[22]	Fill pending
D[21]	Next Victim
D[20]	Valid
D[19:0]	Tag

Each access to the data cache dirty bits read or write the bits for both ways of two sets. Only even addressed sets should be addressed. So the following table shows Set Index bit 0 and Way as always zero.

Table 7-5: Data Cache Dirty Bits extended data access address

Address Bit	Meaning
A[31:20]	L1 Data Block A address
A[19]	PARCTL
A[18]	PARSEL
A[17:11]	1110100
A[10:5]	Set Index bits 7 to 1
A[4:0]	00000

Table 7-6: Data Cache Dirty Bits extended data access value

Data Bit	Value
D[31:8]	0
D[7]	Odd Set Way 1 Dirty Parity
D[6]	Odd Set Way 0 Dirty Parity
D[5]	Odd Set Way 1 Dirty
D[4]	Odd Set Way 0 Dirty
D[3]	Even Set Way 1 Dirty Parity
D[2]	Even Set Way 0 Dirty Parity
D[1]	Even Set Way 1 Dirty
D[0]	Even Set Way 0 Dirty

Cached data is stored in SRAM banks which may be accessed at their native address in ENX mode.

Table 7-7: Data Cache Data extended data access address and mapping to L1 SRAM subbank

L1 SRAM Address Bits	Cache Location
A[31:21]	L1 Data SRAM Address
A[20]	Set Index bit 8 selects Block A or B
A[19]	PARCTL
A[18]	PARSEL
A[17:15]	Usually 0, depends on size of data Block
A[14]	1
A[13]	Set Index bit 7
A[12]	Way
A[11:5]	Set Index bits 6 to 0
A[4:0]	Byte offset in cache line

The 20 bit Tag, 9 bit Set Index, and 5 bit offset within the cache line can be combined to form the home address of data in the cache.

Table 7-8: Data Address from Tag, Set Index, and Offset

L1DM_DCTL.CFG[1]	L1DM_DCTL.DCBS	Address
0	0	Tag[19:12], Tag[11], Tag[10:3], Tag[2], Tag[1], Set[7:0], Offset[4:0]
0	1	Tag[19:12], Tag[2], Tag[10:3], Tag[11], Tag[1], Set[7:0], Offset[4:0]
1	0	Tag[19:12], Tag[11], Tag[10:3], Set[8], Tag[1], Set[7:0], Offset[4:0]
1	1	Tag[19:12], Set[8], Tag[10:3], Tag[11], Tag[1], Set[7:0], Offset[4:0]

The instruction cache has a slightly different format because it has 4 Ways and 128 Sets, unlike the data cache which has 2 Ways and 256 Sets.

Table 7-9: Instruction Cache Tags extended data access address

Address Bit	Meaning
A[31:20]	L1 Instruction SRAM Address
A[19]	PARCTL
A[18]	PARSEL
A[17:13]	11111
A[12:5]	Set Index
A[4:3]	Way
A[2:0]	000

The Instruction Cache Tags extended data access value is the same as for Data Cache.

Table 7-10: Instruction Cache Data extended data access address and mapping to L1 SRAM subbank.

L1 SRAM Address Bits	Cache Location
A[31:20]	L1 Instruction SRAM Address
A[19]	PARCTL
A[18]	PARSEL
A[17:15]	Usually 001, depends on size of L1 instruction SRAM
A[14]	1
A[13:12]	Way
A[11:5]	Set Index
A[4:0]	Byte offset in cache line

Cache Fills and Victims

Cacheable accesses do not use the Read/Write buffer, but a similar structure called the Fill/Victim buffer. These buffers are used on a cache read miss and when a cache read miss generates a victim cache line that needs to be written back to memory. A fill/victim buffer entry consists of an address, 32 bytes of data, and cache line status information.

The CPLBs describe the cache mode to be used for various memory regions. The supported cache modes are Non-cacheable, Write-Back and Write-Through. Traffic for non-cacheable memory are system, or non-speculative accesses, described in System Memory Access.

- In write-through mode a write to external memory is initiated immediately upon the write to cache. If the cache line is present in cache it is also updated there, but the cache line is not fetched on a write miss. If the cache line is replaced or explicitly flushed by software, the contents of the cache line are invalidated rather than written back to external memory.
- In write-back mode the cache does not write to external memory until the line is replaced by a load operation that needs the line. For most applications, a write-back cache is more efficient than a write-through cache, as the external memory accesses are less frequent. On a write miss the cache line is filled from system, and then modified in the cache by the write data.

When a read miss occurs a cache line fill is always initiated.

A cacheable write or a read which misses the cache will allocate a Fill/Victim buffer entry. If a write also causes a cache fill, the write data and fill data will be merged into the Fill buffer before being drained to L1 memory.

When a modified cache line is evicted from the cache, it must be written back to memory. In the reverse process of a fill, a Fill buffer is allocated to receive the cache line from L1 memory, then forward it to system memory. The process of reading the cache victim from L1 requires the DAG interface to be stalled, as the DAG0 read port is used to read the victimized cache line from L1 memory. Cache victims are created by cache flush operations, or when a cache fill displaces a cache line in the same set because the cache set is full.

The Fill/Victim buffer performs data forwarding in the same manner as the Read/Write buffer. If valid data exists in the Fill/Victim buffer, it can be forwarded to the history buffer with few limitations. Write gathering is also done in the Fill/Victim buffer.

The Fill/Victim buffer can stall the L1 memory pipeline if buffers are not available, or if it is in a state where it cannot respond to a request.

System Memory Interface

The Blackfin core memory interface to the system is used to access memory regions outside of L1 and MMR space. The data widths of the read and write data buses are 64 bits.

Cache and non-cache transactions are signaled across the memory interface. Cache transactions are 256 bits, and so result in a burst of 4 64-bit words. The burst type signaled for cache transactions is `WRAP` mode. For cache reads, a core expects the critical word to be the first one returned as signified by the address sent during the read transaction. Non-cache transactions, whether for data or instruction, use a burst type of `INCR` with a transaction length of 1.

The `TESTSET` instruction is supported via a bus-locked transaction. When a `TESTSET` instruction is committed, all pending memory transactions are completed before the read portion of the `TESTSET` is initiated. After the read data is returned, the write portion of the `TESTSET` instruction will be initiated to the same memory location. After a write response is received by the interface, other memory transactions will once again be allowed. If the `TESTSET` instruction is killed in the processor pipe before commit, a dummy write will be performed on the system crossbar with all byte enables set to inactive to clear the lock.

Load and Store Exclusive instructions are supported by exclusive bus transactions.

System Slave Interface

The Blackfin core interface is an SCB slave interface used to access a core's L1 memory. The width of the read and write data buses are 32 bits. This interface is used for DMA access to the L1 memory spaces.

SCB transaction burst lengths of 1 to 16 are supported. All burst types (Fixed address, Incrementing address, and Wrap mode) are supported. All burst sizes less than or equal to the bus width are supported. Locked and exclusive accesses are not supported by the interface. SCB cache information signaling is not supported by the interface. Arbitrary write strobes are supported. The slave interface does not reorder read or write transactions. However, no ordering is enforced between the read and write transaction streams.

The slave interface will respond with a slave error for illegal accesses. During read responses, an error will be reported for every access within the read burst that attempted an illegal access. For write responses, an error will be reported if any access of the burst attempted an illegal access. Illegal accesses do not result in transactions being initiated to the L1 memory.

The following accesses result in a slave error:

- Access to a non-populated L1 region
- Access to an L1 region programmed as cache

System MMR Interface

The Blackfin system MMR interface is a crossbar master interface used to access the system MMR memory region. The data width of the read and write data buses are 32 bits. Only a single system MMR transaction can be active at any time. Therefore, the read and write address buses are shared in the interface. The length of all transactions is 1. Transaction sizes supported are 8, 16 and 32 bits.

All reads via the system MMR interface are non-speculative.

Terminology

The following terminology is used to describe memory.

cache block

The smallest unit of memory that is transferred to/from the next level of memory from/to a cache as a result of a cache miss.

cache hit

A memory access that is satisfied by a valid, present entry in the cache.

cache line

Same as cache block. In this chapter, cache line is used for cache block.

cache miss

A memory access that does not match any valid entry in the cache.

direct-mapped

Cache architecture in which each line has only one place in which it can appear in the cache. Also described as 1-Way associative.

dirty/modified

A state bit, stored along with the tag, indicating whether the data in the data cache line has been changed since it was copied from the source memory and, therefore, needs to be updated in that source memory.

exclusive, clean

The state of a data cache line, indicating that the line is valid and that the data contained in the line matches that in the source memory. The data in a clean cache line does not need to be written to source memory before it is replaced.

fully associative

Cache architecture in which each line can be placed anywhere in the cache.

index

Address portion that is used to select an array element (for example, a line index).

invalid

Describes the state of a cache line. When a cache line is invalid, a cache line match cannot occur.

least recently used (LRU) algorithm

Replacement algorithm, used by cache, that first replaces lines that have been unused for the longest time.

Level 1 (L1) memory

Memory that is directly accessed by the core with no intervening memory subsystems between it and the core.

little endian

The native data store format of the Blackfin processor. Words and half words are stored in memory (and registers) with the least significant byte at the lowest byte address and the most significant byte in the highest byte address of the data storage location.

replacement policy

The function used by the processor to determine which line to replace on a cache miss. Often, an LRU algorithm is employed.

set

A group of *N*-line storage locations in the Ways of an *N*-Way cache, selected by the `INDEX` field of the address (see the Cache Lines figures).

set associative

Cache architecture that limits line placement to a number of sets (or Ways).

tag

Upper address bits, stored along with the cached data line, to identify the specific address source in memory that the cached line represents.

valid

A state bit, stored with the tag, indicating that the corresponding tag and data are current and correct and can be used to satisfy memory access requests.

victim

A dirty cache line that must be written to memory before it can be replaced to free space for a cache line allocation.

Way

An array of line storage elements in an *N*-Way cache (see the Cache Lines figures).

write back

A cache write policy, also known as *copyback*

ADSP-BF70x L1IM Register Descriptions

L1 Instruction Memory Unit (L1IM) contains the following registers.

Table 7-11: ADSP-BF70x L1IM Register List

Name	Description
L1IM_ICTL	Instruction Memory Control Register
L1IM_ISTAT	Instruction Memory CPLB Status Register
L1IM_ICPLB_FAULT_ADDR	Instruction Memory CPLB Fault Address Register
L1IM_ICPLB_DFLT	Instruction Memory CPLB Default Settings Register
L1IM_IPERR_STAT	Instruction Parity Error Status Register
L1IM_ICPLB_ADDRn	Instruction Memory CPLB Address Registers
L1IM_ICPLB_DATAn	Instruction Memory CPLB Data Registers

Instruction Memory Control Register

The L1IM_ICTL register controls memory management unit operation of CPLBs. This register enables CPLB operation, configures memory block usage, and selects the configuration of other CPLB controls.

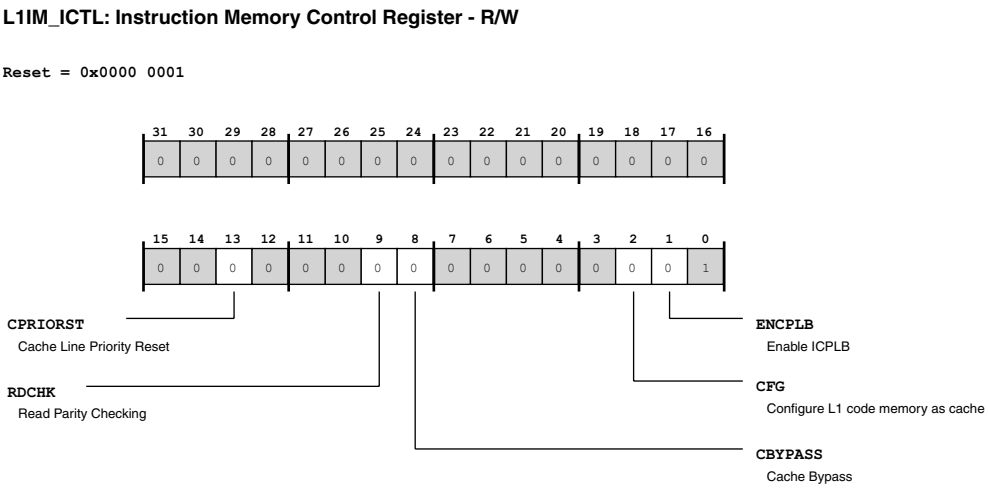


Figure 7-8: L1IM_ICTL Register Diagram

Table 7-12: L1IM_ICTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
13 (R/W)	CPRIORST	Cache Line Priority Reset.	
		0	Normal operation (store LRUPRIO bits in all CPLBs)
		1	Reset (clear/disable) LRUPRIO bits in all CPLBs
9 (R/W)	RDCHK	Read Parity Checking.	
		0	Disable read parity checking
		1	Enable read parity checking
8 (R/W)	CBYPASS	Cache Bypass.	
		0	Disable cache bypass (normal cache behavior)
		1	Enable cache bypass
2 (R/W)	CFG	Configure L1 code memory as cache.	
		0	L1 instruction cache/SRAM block as SRAM
		1	L1 instruction cache/SRAM block as cache
1 (R/W)	ENCPLB	Enable ICPLB.	
		0	Disable CPLB operations
		1	Enable CPLB operation

Instruction Memory CPLB Status Register

The L1IM_ISTAT register holds bits that identify status information regarding the CPLB fault during access to instruction memory. These bits indicate the processor mode during the access, indicate whether or not the access was to an illegal address, and indicate which CPLB entry is associated with the fault. Note that the status in formation in the L1IM_ISTAT register only is valid while the processor is executing the fault exception service routine.

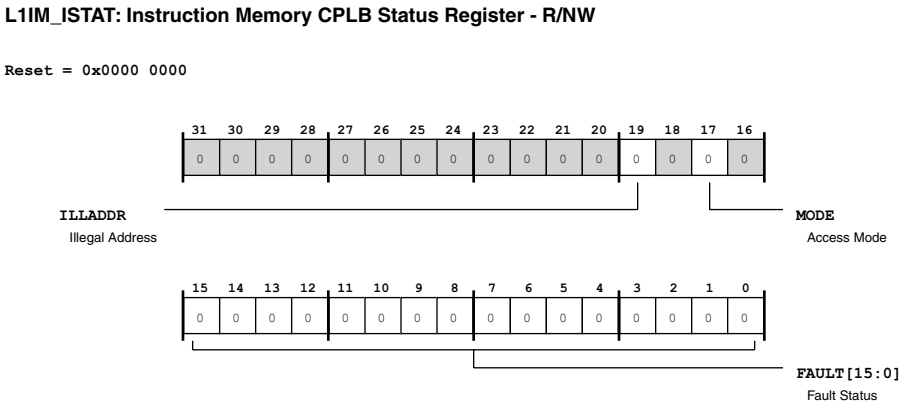


Figure 7-9: L1IM_ISTAT Register Diagram

Table 7-13: L1IM_ISTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
19 (R/NW)	ILLADDR	Illegal Address.	
		0	No status
		1	Illegal address fault
17 (R/NW)	MODE	Access Mode.	
		0	User mode during fault
		1	Supervisor mode during fault
15:0 (R/NW)	FAULT	Fault Status. Each bit indicates the hit/miss status of the associated CPLB entry	

Instruction Memory CPLB Fault Address Register

The L1IM_ICPLB_FAULT_ADDRregister holds the address value of the memory location that caused a fault in L1 instruction memory.

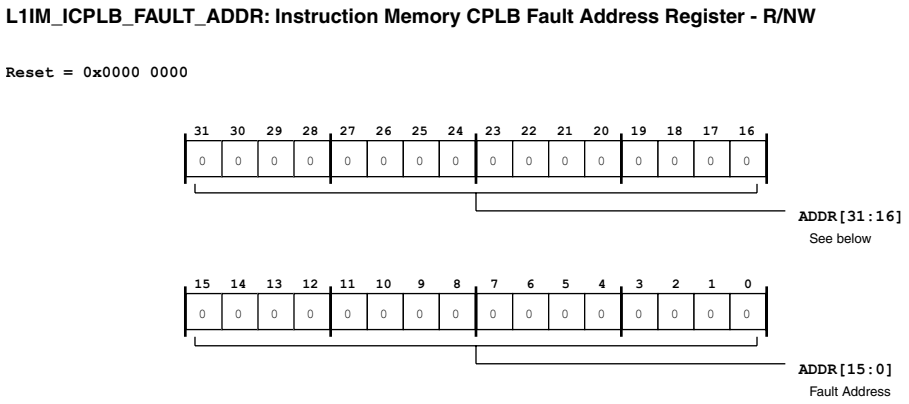


Figure 7-10: L1IM_ICPLB_FAULT_ADDR Register Diagram

Table 7-14: L1IM_ICPLB_FAULT_ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/NW)	ADDR	Fault Address.

Instruction Memory CPLB Default Settings Register

The L1IM_ICPLB_DFLT register selects the default CPLB settings for new instruction memory CPLB entries. These default setting may be changed by writing the CPLB descriptor in the L1IM_ICPLB_DATAn register after the new entry is added.

L1IM_ICPLB_DFLT: Instruction Memory CPLB Default Settings Register - R/W

Reset = 0x0000 0000

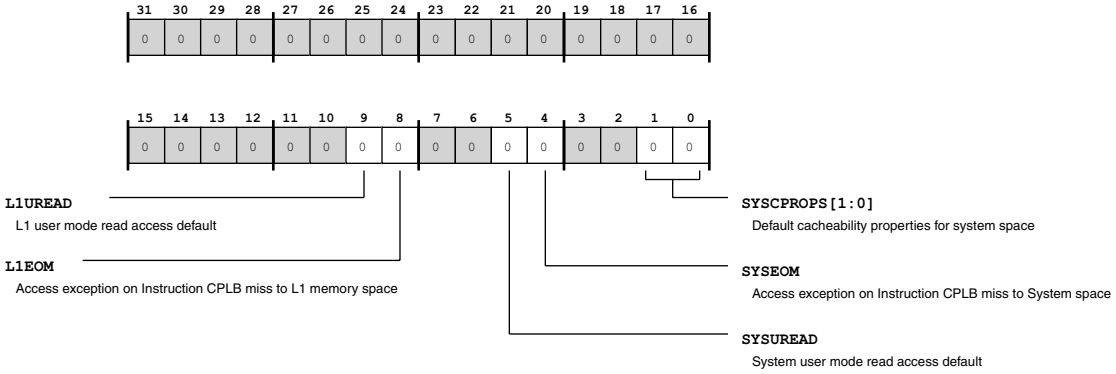


Figure 7-11: L1IM_ICPLB_DFLT Register Diagram

Table 7-15: L1IM_ICPLB_DFLT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
9 (R/W)	L1UREAD	L1 user mode read access default.	
		0	No read access permitted in user mode
		1	Permit read access in user mode
8 (R/W)	L1EOM	Access exception on Instruction CPLB miss to L1 memory space.	
		0	Normal operation (generate exception on CPLB miss)
		1	Disable exception generation on CPLB miss
5 (R/W)	SYSUREAD	System user mode read access default.	
		0	No read access permitted in user mode
		1	Permit read access in user mode
4 (R/W)	SYSEOM	Access exception on Instruction CPLB miss to System space.	
		0	Normal operation (generate exception on CPLB miss)
		1	Disable exception generation on CPLB miss

Table 7-15: L1IM_ICPLB_DFLT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
1:0 (R/W)	SYSCPROPS	Default cacheability properties for system space.	
		0	Non-cacheable memory space
		1	Non-cacheable in L2; cacheable in L1
		2	Cacheable in L2; non-cacheable in L1
		3	Cacheable in L1 and L2

Instruction Parity Error Status Register

The L1IM_IPERR_STAT register contains status information for identifying the location and properties of a parity error occurring during a read access of an address in L1 memory.

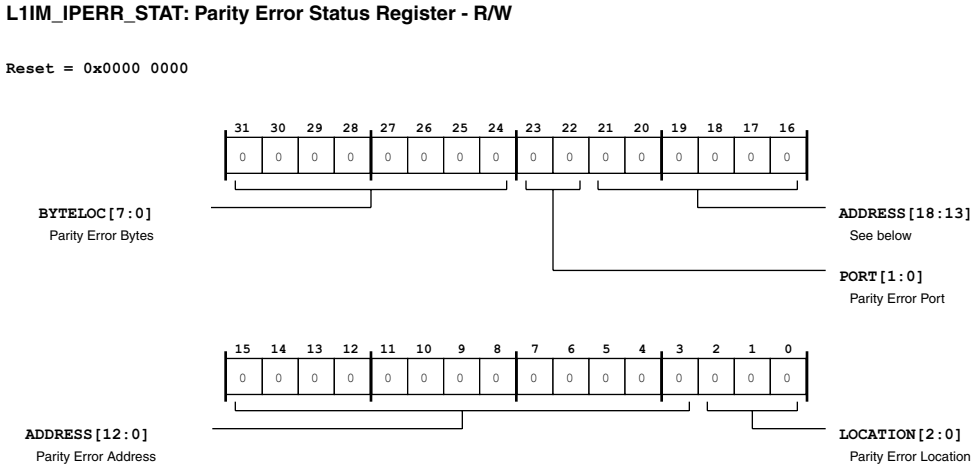


Figure 7-12: L1IM_IPERR_STAT Register Diagram

Table 7-16: L1IM_IPERR_STAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:24 (R/W1C)	BYTELOC	Parity Error Bytes.

Table 7-16: L1IM_IPERR_STAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
23:22 (R/NW)	PORT	Parity Error Port. Specifies read port on which the parity error occurred.	
		0	Port 0 parity error Parity Error source is Port0
		1	Reserved Parity Error source is Port1
		2	DMA parity error Parity Error source is DMA
		3	Reserved Parity Error source is Cache Victim
21:3 (R/NW)	ADDRESS	Parity Error Address.	
2:0 (R/W1C)	LOCATION	Parity Error Location.	
		0	L1 SRAM contains parity error
		01x	Reserved
		1	Tag 0 memory contains parity error
		1xx	Reserved

Instruction Memory CPLB Address Registers

The `L1IM_ICPLB_ADDRn` registers hold descriptors (half of a CPLB entry) for the instruction cacheability protection lookaside buffers.

NOTE: To ensure proper behavior and future compatibility, all reserved bits in this register must be cleared (=0) whenever this register is written.

Each CPLB entry consists of a pair of 32-bit values. For instruction fetch operations, `L1IM_ICPLB_ADDRn` defines the start address of the page described by the CPLB descriptor, and `L1IM_ICPLB_DATAn` defines the properties of the page described by the CPLB descriptor.

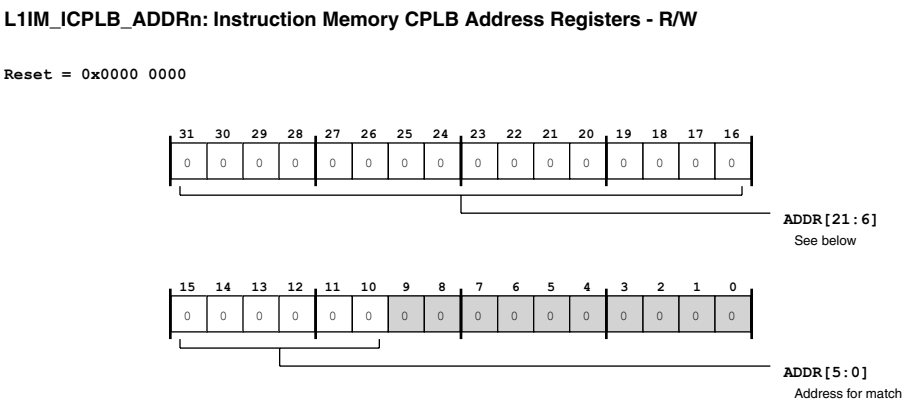


Figure 7-13: L1IM_ICPLB_ADDRn Register Diagram

Table 7-17: L1IM_ICPLB_ADDRn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:10 (R/W)	ADDR	Address for match.

Instruction Memory CPLB Data Registers

The L1IM_ICPLB_DATAn registers hold descriptors (half of a CPLB entry) for the instruction cacheability protection lookaside buffers.

NOTE: To ensure proper behavior and future compatibility, all reserved bits in this register must be cleared (=0) whenever this register is written.

Each CPLB entry consists of a pair of 32-bit values. For instruction fetch operations, L1IM_ICPLB_ADDRn defines the start address of the page described by the CPLB descriptor, and L1IM_ICPLB_DATAn defines the properties of the page described by the CPLB descriptor.

L1IM_ICPLB_DATAn: Instruction Memory CPLB Data Registers - R/W

Reset = 0x0000 0000

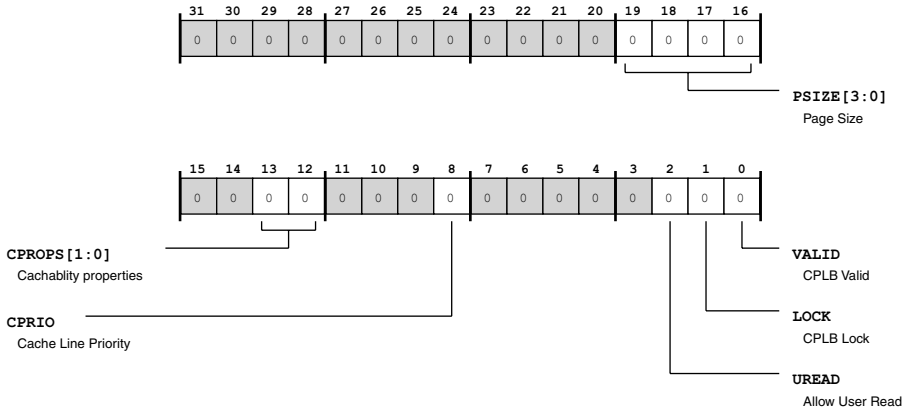


Figure 7-14: L1IM_ICPLB_DATAn Register Diagram

Table 7-18: L1IM_ICPLB_DATAn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
19:16 (R/W)	PSIZE	Page Size. Page size is given by the formula: page size = 4^(PSize+5).	
		0	1K byte page size
		1	4K byte page size
		2	16K byte page size
		3	64K byte page size
		4	256K byte page size
		5	1M byte page size
		6	4M byte page size
		7	16M byte page size
		8	64M byte page size
		9	256M byte page size
		10	1G byte page size
13:12 (R/W)	CPROPS	Cacheability properties.	
		0	Non-cacheable memory space
		1	Non-cacheable in L2; cacheable in L1
		2	Cacheable in L2; non-cacheable in L1
		3	Cacheable in L1 and L2

Table 7-18: L1IM_ICPLB_DATAn Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
8 (R/W)	CPRIO	Cache Line Priority.	
		0	Low importance
		1	High importance
2 (R/W)	UREAD	Allow User Read.	
		0	No read access permitted in user mode
		1	Permit read access in user mode
1 (R/W)	LOCK	CPLB Lock.	
		0	Unlocked - CPLB entry can be replaced
		1	Locked - CPLB entry should not be replaced
0 (R/W)	VALID	CPLB Valid.	
		0	Invalid entry
		1	Valid entry

ADSP-BF70x L1DM Register Descriptions

L1 Data Memory Unit (L1DM) contains the following registers.

Table 7-19: ADSP-BF70x L1DM Register List

Name	Description
L1DM_SRAM_BASE_ADDR	SRAM Base Address Register
L1DM_DCTL	Data Memory Control Register
L1DM_DSTAT	Data Memory CPLB Status Register
L1DM_DCPLB_FAULT_ADDR	Data Memory CPLB Fault Address Register
L1DM_DCPLB_DFLT	Data Memory CPLB Default Settings Register
L1DM_DPERR_STAT	Data Memory Parity Error Status Register
L1DM_DCPLB_ADDRn	Data Memory CPLB Address Registers
L1DM_DCPLB_DATAn	Data Memory CPLB Data Registers

SRAM Base Address Register

When the data or instruction memories are configured as SRAM (see the L1DM_DCTL register description and the L1_IM_ICTL register description), the base address is determined from the L1DM_SRAM_BASE_ADDR register. The SRAM base address inputs to the core are latched into this register at reset. The SRAM base address is aligned to a 4M byte boundary. The SRAM base address cannot overlap the uppermost 4M byte region, because this region is reserved for the processor core and chip-level memory mapped registers. The L1DM_SRAM_BASE_ADDR register accessible only in the supervisor mode.

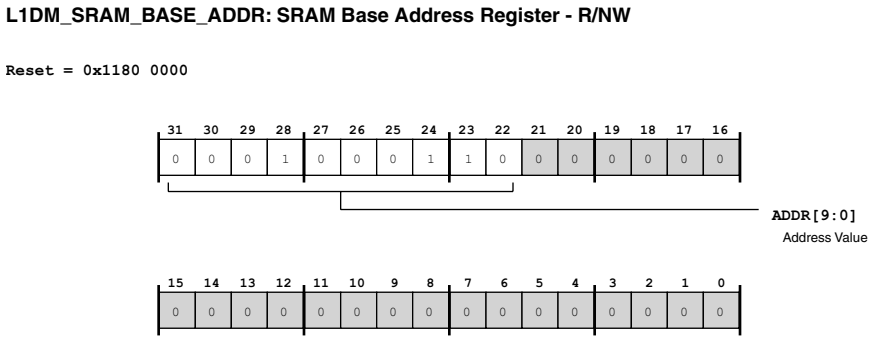


Figure 7-15: L1DM_SRAM_BASE_ADDR Register Diagram

Table 7-20: L1DM_SRAM_BASE_ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:22 (R/NW)	ADDR	Address Value. The L1DM_SRAM_BASE_ADDR.ADDR bits hold the address value for the SRAM base address. This address is aligned to any 4M byte boundary, except for the uppermost 4M byte region. This region is reserved for memory mapped registers.

Data Memory Control Register

The L1DM_DCTL register controls memory management unit operation of CPLBs. This register enables CPLB operation, configures memory block usage, and selects the configuration of other CPLB controls.

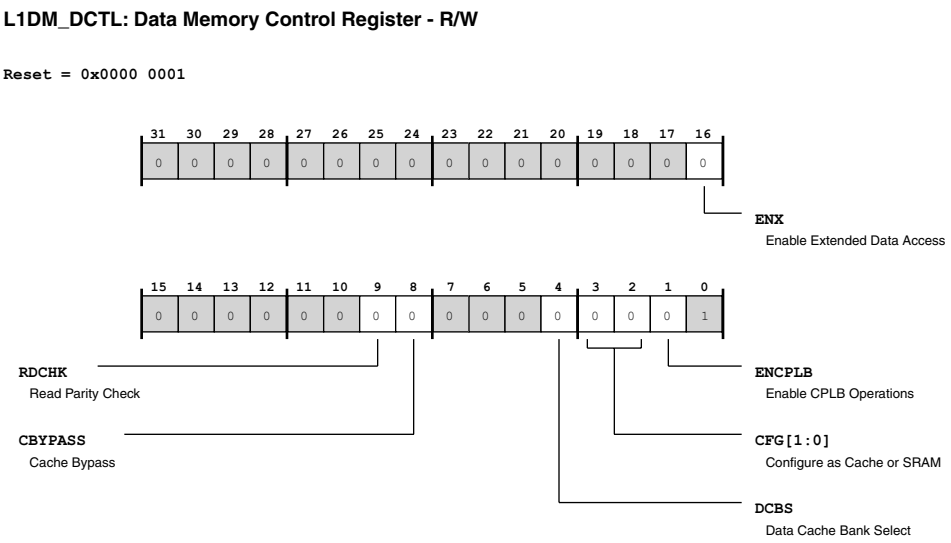


Figure 7-16: L1DM_DCTL Register Diagram

Table 7-21: L1DM_DCTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
16 (R/W)	ENX	Enable Extended Data Access. The L1DM_DCTL.ENX bit enables extended data access mode. When extended data access mode is enabled, all restricted memory spaces (including L1 instruction memory) may be directly accessed by suitably privileged software. For more information, see the description of this mode in the Extended Data Access section.	
		0	Disable extended access (normal operation)
		1	Enable extended access
9 (R/W)	RDCHK	Read Parity Check. The L1DM_DCTL.RDCHK bit enables parity checking for read accesses.	
		0	Disable read parity checking
		1	Enable read parity checking
8 (R/W)	CBYPASS	Cache Bypass. The L1DM_DCTL.CBYPASS bit enables cache bypass, disabling processor usage of the data cache.	
		0	Disable cache bypass (normal cache behavior)
		1	Enable cache bypass

Table 7-21: L1DM_DCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
4 (R/W)	DCBS	Data Cache Bank Select. The L1DM_DCTL.DCBS bit selects whether the memory management unit uses address bit 14 or 23 to choose L1 data memory bank A or B.	
		0	Bit 14 used to select L1 data bank A or B If bit 14 of address =1, select L1 data memory data bank A; if bit 14 of address is 0, select L1 data memory data bank B.
		1	Bit 23 used to select L1 data bank A or B If bit 23 of address =1, select L1 data memory data bank A; if bit 23 of address is 0, select L1 data memory data bank B.
3:2 (R/W)	CFG	Configure as Cache or SRAM. The L1DM_DCTL.CFG bits configure the usage of L1 memory data blocks as cache or SRAM. When this field is cleared (=0), this write invalidates all cache lines if previously configured as cache.	
		0	L1 data block A as SRAM, L1 data block B as SRAM Also invalidates all cache lines if previously configured as cache
		1	L1 data block A as cache, L1 data block B as SRAM
		3	L1 data block A as cache, L1 data block B as cache
1 (R/W)	ENCPLB	Enable CPLB Operations. The L1DM_DCTL.ENCPLB bit enables data memory CPLB operations. When disabled, the memory management unit only performs minimal address checking.	
		0	Disable CPLB operations
		1	Enable CPLB operations

Data Memory CPLB Status Register

The L1DM_DSTAT register holds bits that identify status information regarding the CPLB fault during access to data memory. These bits indicate the processor mode during the access, indicated whether the access was read or write, indicate whether it was a DAG access, indicate whether or not the access was to an illegal address, and indicate which CPLB entry is associated with the fault. Note that the status in formation in the L1DM_DSTAT register only is valid while the processor is executing the fault exception service routine.

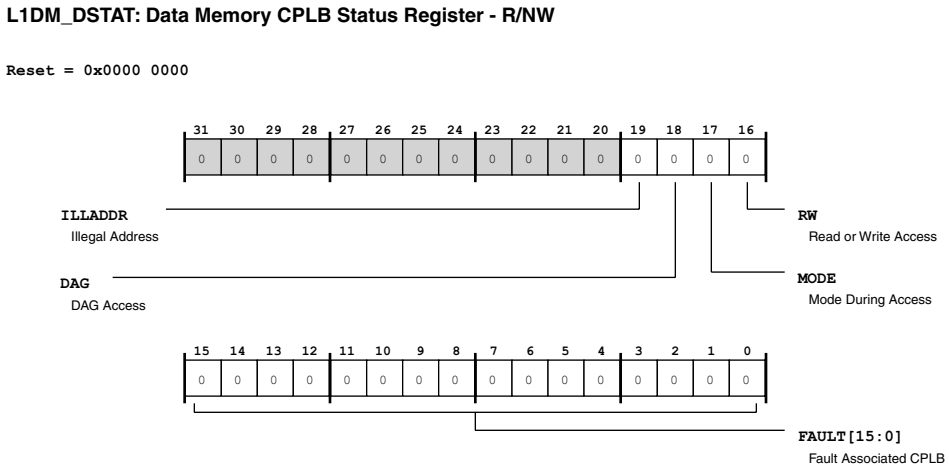


Figure 7-17: L1DM_DSTAT Register Diagram

Table 7-22: L1DM_DSTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
19 (R/NW)	ILLADDR	Illegal Address. The L1DM_DSTAT.ILLADDR bit indicates whether the fault occurred through an attempt to access non-existent memory.	
		0	No status
		1	Illegal address fault
18 (R/NW)	DAG	DAG Access. The L1DM_DSTAT.DAG bit indicates whether the fault access was made by DAG 0 or 1.	
		0	DAG 0 access fault
		1	DAG 1 access fault
17 (R/NW)	MODE	Mode During Access. The L1DM_DSTAT.MODE bit indicates whether the processor mode was user or supervisor during the fault access.	
		0	User mode during fault
		1	Supervisor mode during fault
16 (R/NW)	RW	Read or Write Access. The L1DM_DSTAT.RW bit indicates whether the fault access was a read or write access.	
		0	Read access fault
		1	Write access fault
15:0 (R/NW)	FAULT	Fault Associated CPLB. The L1DM_DSTAT.FAULT bits each indicate the fault status of the associated CPLB entry. (Fault =1; No status =0)	

Data Memory CPLB Fault Address Register

The L1DM_DCPLB_FAULT_ADDR register holds the address value of the memory location that caused a fault in L1 data memory.

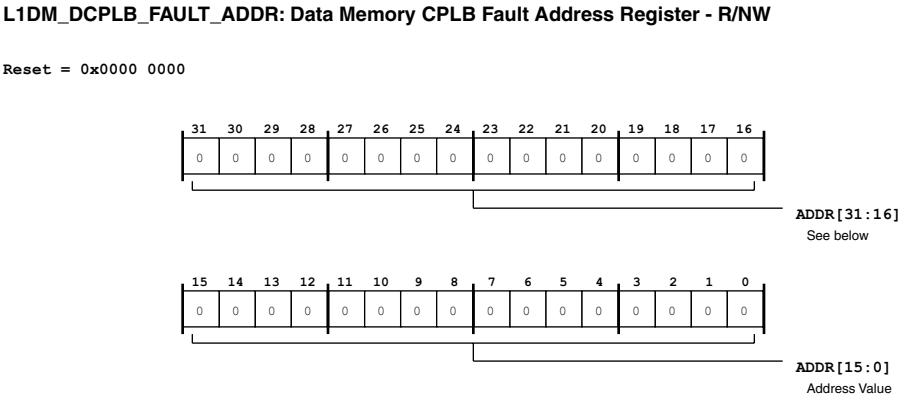


Figure 7-18: L1DM_DCPLB_FAULT_ADDR Register Diagram

Table 7-23: L1DM_DCPLB_FAULT_ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/NW)	ADDR	Address Value. The L1DM_DCPLB_FAULT_ADDR.ADDRbits hold the address value of the memory location that caused a fault in L1 data memory.

Data Memory CPLB Default Settings Register

The L1DM_DCPLB_DFLT register selects the default CPLB settings for new data memory CPLB entries. These default setting may be changed by writing the CPLB descriptor in the L1DM_DCPLB_DATAn register after the new entry is added.

L1DM_DCPLB_DFLT: Data Memory CPLB Default Settings Register - R/W

Reset = 0x0000 0000

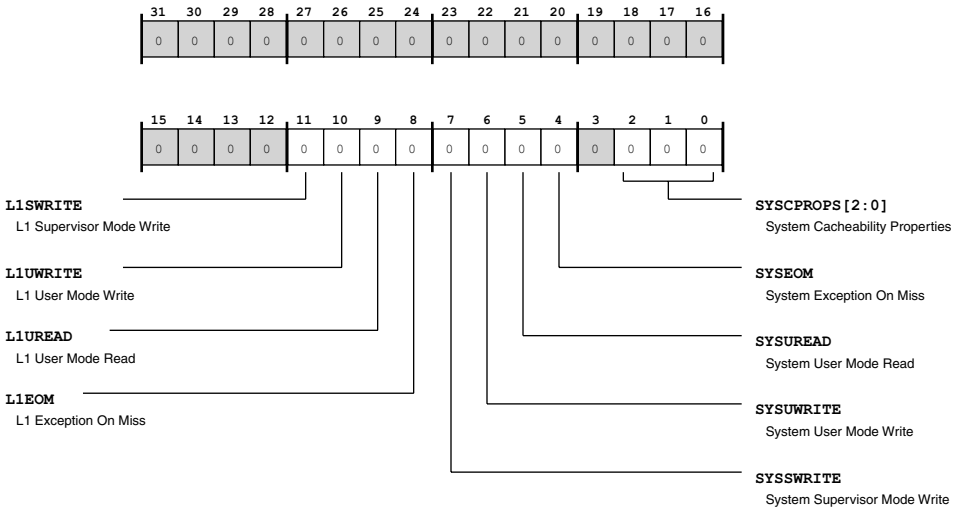


Figure 7-19: L1DM_DCPLB_DFLT Register Diagram

Table 7-24: L1DM_DCPLB_DFLT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
11 (R/W)	L1SWRITE	L1 Supervisor Mode Write. The L1DM_DCPLB_DFLT.L1SWRITE bit selects the default for supervisor mode write access to L1 memory. This selection determines whether the default setting for each page is to permit or restrict supervisor mode write access. If access is made when restricted, the access generates a protection violation exception. When the memory management unit adds a CPLB entry, the L1DM_DCPLB_DFLT.L1SWRITE selection provides the default setting for the page. After the entry is added, this selection may be changed by writing to the L1DM_DCPLB_DATAn.SWRITE bit.	
		0	No write access permitted in supervisor mode
		1	Permit write access in supervisor mode
10 (R/W)	L1UWRITE	L1 User Mode Write. The L1DM_DCPLB_DFLT.L1UWRITE bit selects the default for user mode write access to L1 memory. This selection determines whether the default setting for each page is to permit or restrict user mode write access. If access is made when restricted, the access generates a protection violation exception. When the memory management unit adds a CPLB entry, the L1DM_DCPLB_DFLT.L1UWRITE selection provides the default setting for the page. After the entry is added, this selection may be changed by writing to the L1DM_DCPLB_DATAn.UWRITE bit.	
		0	No write access permitted in user mode
		1	Permit write access in user mode

Table 7-24: L1DM_DCPLB_DFLT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
9 (R/W)	L1UREAD	L1 User Mode Read. The L1DM_DCPLB_DFLT.L1UREAD bit selects the default for user mode read access to L1 memory. This selection determines whether the default setting for each page is to permit or restrict user mode read access. If access is made when restricted, the access generates a protection violation exception. When the memory management unit adds a CPLB entry, the L1DM_DCPLB_DFLT.L1UREAD selection provides the default setting for the page. After the entry is added, this selection may be changed by writing to the L1DM_DCPLB_DATAn.UREAD bit.	
		0	No read access permitted in user mode
		1	Permit read access in user mode
8 (R/W)	L1EOM	L1 Exception On Miss. The L1DM_DCPLB_DFLT.L1EOM bit disables access exception generation on DAG CPLB miss to L1 memory space. Default access protection of L1 memory space is controlled by the L1DM_DCPLB_DFLT.L1UREAD, L1DM_DCPLB_DFLT.L1UWRITE and L1DM_DCPLB_DFLT.L1SWRITE bits.	
		0	Normal operation (generate exception on CPLB miss)
		1	Disable exception generation on CPLB miss
7 (R/W)	SYSSWRITE	System Supervisor Mode Write. The L1DM_DCPLB_DFLT.SYSSWRITE bit selects the default for supervisor mode write access to system space. This selection determines whether the default setting for each page is to permit or restrict supervisor mode write access. If access is made when restricted, the access generates a protection violation exception. When the memory management unit adds a CPLB entry, the L1DM_DCPLB_DFLT.SYSSWRITE selection provides the default setting for the page. After the entry is added, this selection may be changed by writing to the L1DM_DCPLB_DATAn.SWRITE bit.	
		0	No write access permitted in supervisor mode
		1	Permit write access in supervisor mode
6 (R/W)	SYSUWRITE	System User Mode Write. The L1DM_DCPLB_DFLT.SYSUWRITE bit selects the default for user mode write access to system space. This selection determines whether the default setting for each page is to permit or restrict user mode write access. If access is made when restricted, the access generates a protection violation exception. When the memory management unit adds a CPLB entry, the L1DM_DCPLB_DFLT.SYSUWRITE selection provides the default setting for the page. After the entry is added, this selection may be changed by writing to the L1DM_DCPLB_DATAn.UWRITE bit.	
		0	No write access permitted in user mode
		1	Permit write access in user mode

Table 7-24: L1DM_DCPLB_DFLT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
5 (R/W)	SYSUREAD	System User Mode Read. The L1DM_DCPLB_DFLT.SYSUREAD bit selects the default for user mode read access to system space. This selection determines whether the default setting for each page is to permit or restrict user mode read access. If access is made when restricted, the access generates a protection violation exception. When the memory management unit adds a CPLB entry, the L1DM_DCPLB_DFLT.SYSUREAD selection provides the default setting for the page. After the entry is added, this selection may be changed by writing to the L1DM_DCPLB_DATAn.UREAD bit.	
		0	No read access permitted in user mode
		1	Permit read access in user mode
4 (R/W)	SYSEOM	System Exception On Miss. The L1DM_DCPLB_DFLT.SYSEOM bit disables access exception generation on DAG CPLB miss to system space. Default access protection of system space is controlled by the L1DM_DCPLB_DFLT.SYSCPROPS, L1DM_DCPLB_DFLT.SYSUREAD, L1DM_DCPLB_DFLT.SYSUWRITE and L1DM_DCPLB_DFLT.SYSSWRITE bits.	
		0	Normal operation (generate exception on CPLB miss)
		1	Disable exception generation on CPLB miss
2:0 (R/W)	SYSCPROPS	System Cacheability Properties. The L1DM_DCPLB_DFLT.SYSCPROPS bits select the system default cacheability properties pages. This selection determines whether the default setting for each page is non-cacheable or cacheable in L1 or L2 memory. When the memory management unit adds a CPLB entry, the L1DM_DCPLB_DFLT.SYSCPROPS selection provides the default setting for the page. After the entry is added, these selections may be changed by writing to the L1DM_DCPLB_DATAn.CPROPS bits.	
		0	Non-cacheable memory space
		1	Non-cacheable in L2; write back cacheable in L1
		2	Write back cacheable in L2; non-cacheable in L1
		3	Write back cacheable in L1 and L2
		4	I/O device space
		5	Non-cacheable in L2; write through cacheable in L1
		6	Write through cacheable in L2; non-cacheable in L1
		7	Write through cacheable in L1 and L2

Data Memory Parity Error Status Register

The L1DM_DPERR_STAT register contains status information for identifying the location and properties of a parity error occurring during a read access of an address in L1 memory.

L1DM_DPERR_STAT: Data Memory Parity Error Status Register - R/NW

Reset = 0x0000 0000

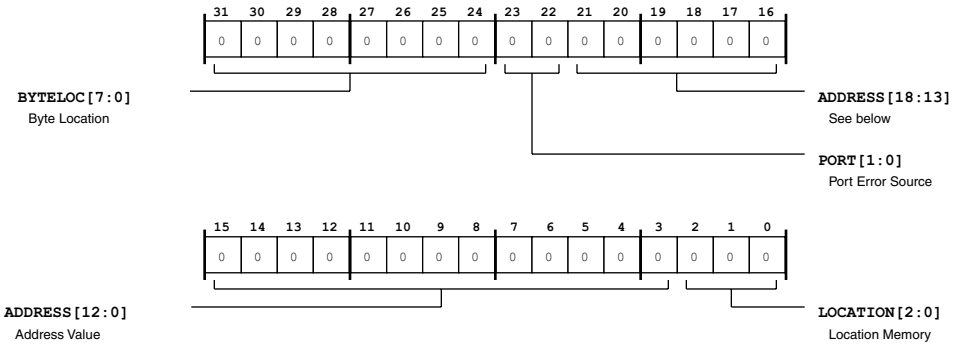


Figure 7-20: L1DM_DPERR_STAT Register Diagram

Table 7-25: L1DM_DPERR_STAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:24 (R/W1C)	BYTELOC	Byte Location. The L1DM_DPERR_STAT.BYTELOC bits indicate the location of bytes in the most recent read, which produced parity error faults. The error indication for bits in this field is quad-word aligned. For example, if a byte access to address 0xFF800005 generates a parity error, the L1DM_DPERR_STAT.BYTELOC[5] bit is set (=1), not the L1DM_DPERR_STAT.BYTELOC[0] bit. Note that these bits are sticky, requiring a W1C action.	
23:22 (R/NW)	PORT	Port Error Source. The L1DM_DPERR_STAT.PORT bits indicate on which read port the parity error occurred.	
		0	Port 0 parity error
		1	Port 1 parity error
		2	DMA parity error
		3	Cache victim parity error
21:3 (R/NW)	ADDRESS	Address Value. The L1DM_DPERR_STAT.ADDRESS bits provide the byte address value of the memory location in the most recent read, which produced the parity error. Note that the L1DM_DPERR_STAT.ADDRESS address value is NOT the same thing as the L1DM_DPERR_STAT.BYTELOC (location of the faulting byte).	

Table 7-25: L1DM_DPERR_STAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
2:0 (R/W1C)	LOCATION	Location Memory. The L1DM_DPERR_STAT.LOCATION bits indicate the memory type containing the location associated with the parity error. Note that these bits are sticky, requiring a W1C action.	
		0	L1 SRAM contains parity error
		1	Tag 0 memory contains parity error
		2	Tag 1 memory contains parity error
		4	Dirty memory contains parity error

Data Memory CPLB Address Registers

The L1DM_DCPLB_ADDRn registers hold descriptors (half of a CPLB entry) for the data cacheability protection lookaside buffers.

NOTE: To ensure proper behavior and future compatibility, all reserved bits in this register must be cleared (=0) whenever this register is written.

Each CPLB entry consists of a pair of 32-bit values. For data operations, L1DM_DCPLB_ADDRn defines the start address of the page described by the CPLB descriptor, and L1DM_DCPLB_DATAn defines the properties of the page described by the CPLB descriptor.

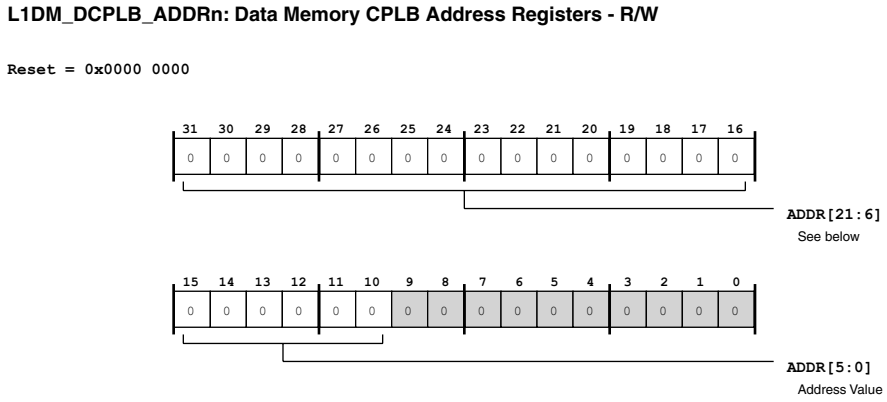


Figure 7-21: L1DM_DCPLB_ADDRn Register Diagram

Table 7-26: L1DM_DCPLB_ADDRn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:10 (R/W)	ADDR	Address Value. The L1DM_DCPLB_ADDRn.ADDR bits contain the upper bits of the address value for match.

Data Memory CPLB Data Registers

The L1DM_DCPLB_DATAn registers hold descriptors (half of a CPLB entry) for the data cacheability protection lookaside buffers.

NOTE: To ensure proper behavior and future compatibility, all reserved bits in this register must be cleared (=0) whenever this register is written.

Each CPLB entry consists of a pair of 32-bit values. For data operations, L1DM_DCPLB_ADDRn defines the start address of the page described by the CPLB descriptor, and L1DM_DCPLB_DATAn defines the properties of the page described by the CPLB descriptor.

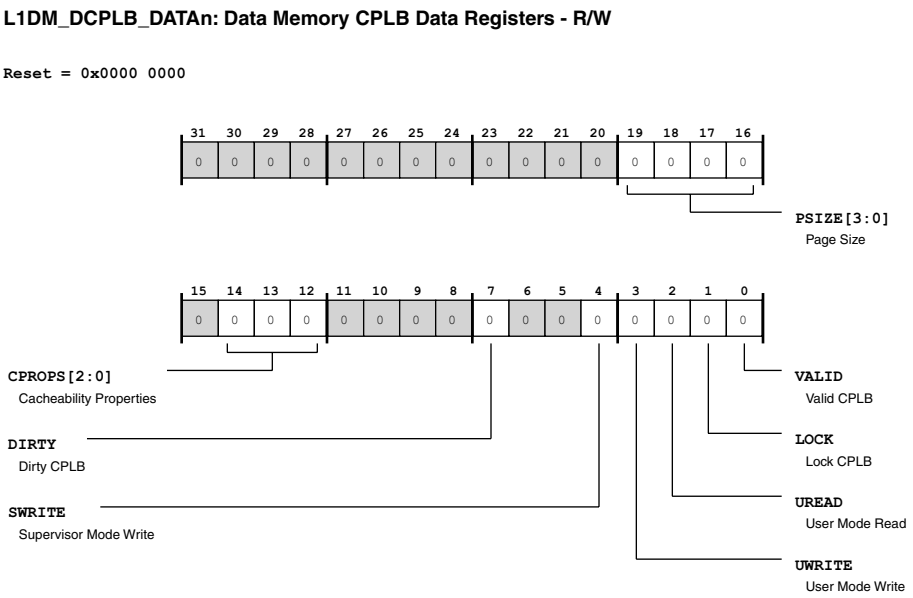


Figure 7-22: L1DM_DCPLB_DATAn Register Diagram

Table 7-27: L1DM_DCPLB_DATAn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
19:16 (R/W)	PSIZE	Page Size. The L1DM_DCPLB_DATAn.PSIZE bits select the page size according to the formula: page size = $4^{(L1DM\_DCPLB\_DATAn.PSIZE+5)}$. Enumeration values not shown are reserved.	
		0	1K byte page size
		1	4K byte page size
		2	16K byte page size
		3	64K byte page size
		4	256K byte page size
		5	1M byte page size
		6	4M byte page size
		7	16M byte page size
		8	64M byte page size
		9	256M byte page size
		10	1G byte page size
14:12 (R/W)	CPROPS	Cacheability Properties. The L1DM_DCPLB_DATAn.CPROPS bits select the cacheability properties for the page. This selection determines whether the page is non-cacheable or cacheable for write back/through in L1 or L2 memory.	
		0	Non-cacheable memory space
		1	Non-cacheable in L2; write back cacheable in L1
		2	Write back cacheable in L2; non-cacheable in L1
		3	Write back cacheable in L1 and L2
		4	I/O device space
		5	Non-cacheable in L2; write through cacheable in L1
		6	Write through cacheable in L2; non-cacheable in L1
		7	Write through cacheable in L1 and L2

Table 7-27: L1DM_DCPLB_DATAn Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
7 (R/W)	DIRTY	Dirty CPLB. The L1DM_DCPLB_DATAn.DIRTY bit is used by software to indicate that a write has been made to the page since the CPLB entry was installed. On the first write to the page after its installation, the MMU exception handler raises a CPLB dirty exception and sets this bit (=1). To avoid this exception, software may set this bit when the CPLB entry is installed. Software may use the L1DM_DCPLB_DATAn.DIRTY bit to detect whether a write as been made to a page, so the write may be propagated to further levels of the memory hierarchy.	
		0	Clean - CPLB dirty exception raised on page write A CPLB dirty exception is raised if the page is written to.
		1	Dirty - No CPLB dirty exception raised on page write A CPLB dirty exception is not raised when the page is written to.
4 (R/W)	SWRITE	Supervisor Mode Write. The L1DM_DCPLB_DATAn.SWRITE bit selects whether to permit or restrict supervisor mode write access. If access is made when restricted, the access generates a protection violation exception.	
		0	No write access permitted in supervisor mode
		1	Permit write access in supervisor mode
3 (R/W)	UWRITE	User Mode Write. The L1DM_DCPLB_DATAn.UWRITE bit selects whether to permit or restrict user mode write access. If access is made when restricted, the access generates a protection violation exception.	
		0	No write access permitted in user mode
		1	Permit write access in user mode
2 (R/W)	UREAD	User Mode Read. The L1DM_DCPLB_DATAn.UREAD bit selects whether to permit or restrict user mode read access. If access is made when restricted, the access generates a protection violation exception.	
		0	No read access permitted in user mode
		1	Permit read access in user mode
1 (R/W)	LOCK	Lock CPLB. The L1DM_DCPLB_DATAn.LOCK bit locks or unlocks the CPLB entry. If locked, the memory management unit is directed to keep this entry in MMR and do not permit this entry to participate in the CPLB replacement policy.	
		0	Unlocked - CPLB entry can be replaced
		1	Locked - CPLB entry should not be replaced

Table 7-27: L1DM_DCPLB_DATAn Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
0 (R/W)	VALID	Valid CPLB. The L1DM_DCPLB_DATAn.VALID bit indicates that the CPLB entry contains valid data. Software uses this bit to identify valid CPLB entries during operation of the MMU exception handler and execution of the CPLB replacement policy.	
		0	Invalid entry
		1	Valid entry

8 Instruction Reference Pages

The instruction reference pages provide detailed information about the syntax and operation of each instruction in the processor's instruction set. The reference groups the instructions by type and by operation. This grouping stems from the portion of the processor core (see the **Enhanced Blackfin Core Block Diagram** figure) on which each instruction executes. Because each instruction uses specific resources (portions of the processor architecture), understanding the relationship between the instructions and the architecture can greatly influence how you write efficient code and achieve optimum code density (applying instruction parallelism).

- Arithmetic Instructions -- execute within the data arithmetic unit
- Sequencer Instructions -- execute within the control unit
- Memory or Pointer Instructions -- execute within the address arithmetic unit
- Specialized Compute Instructions -- execute within the data arithmetic unit

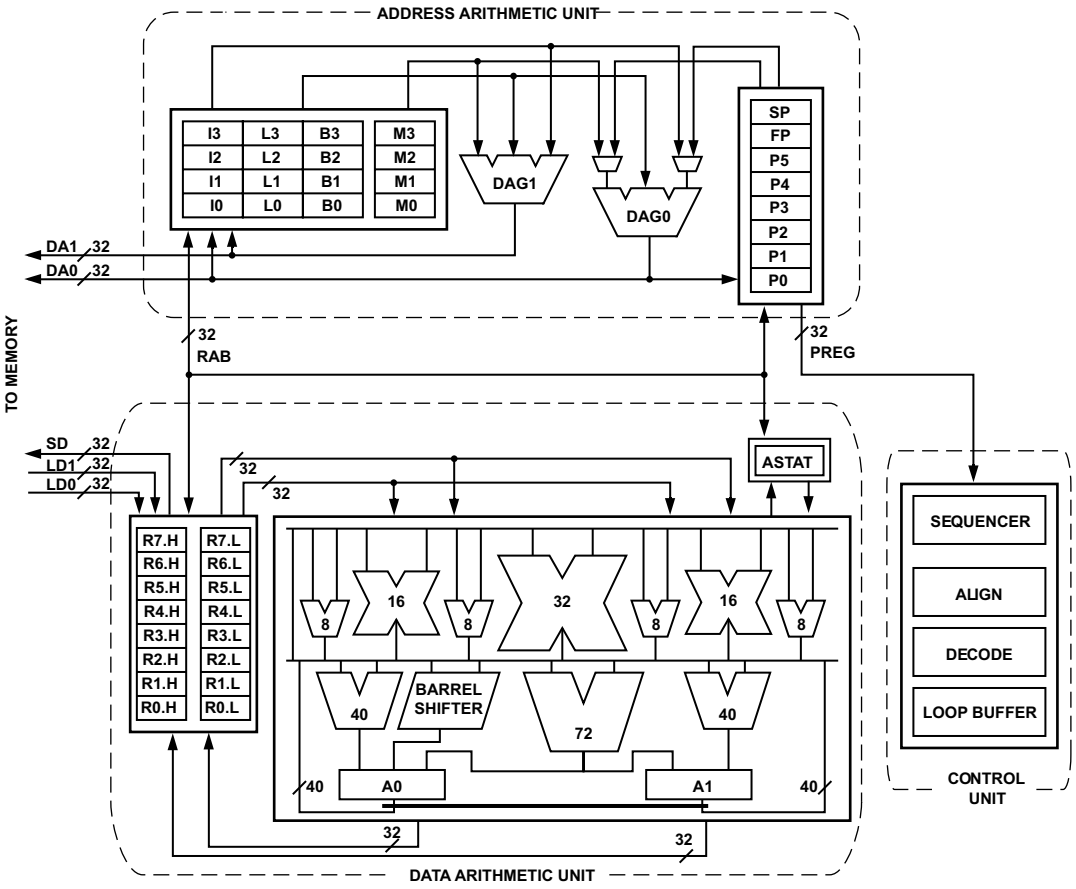


Figure 8-1: Enhanced Blackfin Core Block Diagram

NOTE: Arithmetic instructions generate status, indicating information about the result of the operation. For more information, see Arithmetic Status Register. To optimize program execution, many 16- and 32-bit instructions may be issued in parallel. For more information, see Issuing Parallel Instructions.

For more information about ADSP-BF70x processor family core architecture or memory infrastructure, see the corresponding chapters of this text. For information about ADSP-BF70x processor peripherals, see the hardware reference for the specific processor.

Each instruction reference page provides the following information:

- **Syntax** -- Each section of a syntax table identifies the underlying instruction encoding (for example, ALU Operations (Dsp32Alu)). Each line of a syntax table defines the permitted processor resource classes (for example, a register type) that is permitted in that syntax position. To see the list of resources in a particular class, follow the link for that resource class from the syntax line (for example, DDST0_HL).
- **Data Flow** -- For instructions with sophisticated data placement options, a data flow diagram is provided. These are not provided for all instructions.

- **Abstract** -- The abstract is a brief (1-2 sentence) description of the instruction.
- **Description** -- The description provides a full description of the instruction, including execution options, instruction encoding size, instruction parallelism (if applicable), any special applications, and affect on status flags (if applicable).
- **ASTAT Flags** -- The status flags table (if applicable) provides a graphical overview of the status flag affected by the instructions execution. For instructions that do not affect status, this section is omitted.
- **Example** -- The code examples provide a code snippet that demonstrates the instruction and its options.

Arithmetic Instructions

The arithmetic instructions provide operations, which execute on the **data arithmetic unit** in the processor core. Users can take advantage of these instructions to add, subtract, divide, and multiply, as well as to calculate and store absolute values, detect exponents, round, saturate, and return the number of sign bits.

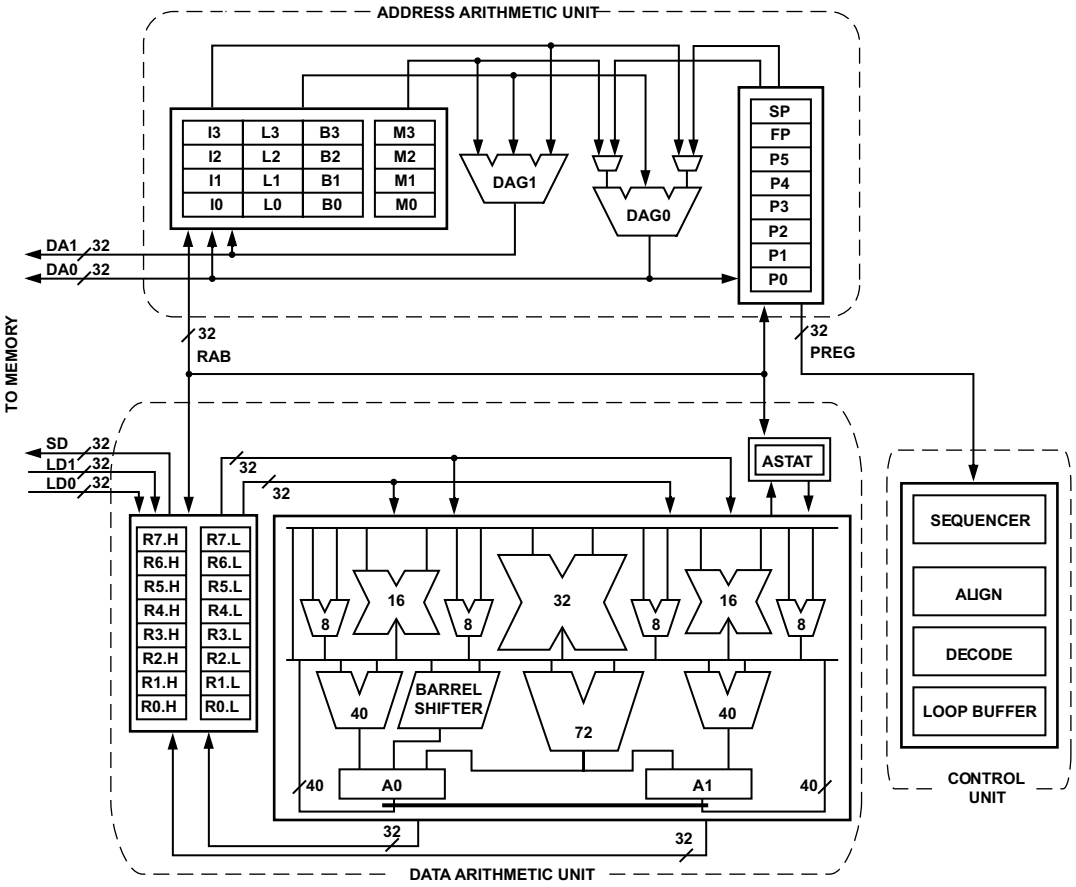


Figure 8-2: Enhanced Blackfin Core Block Diagram

The operation types of arithmetic instructions include:

- Add and Subtract Operations
- Bit Operations
- Comparison Operations
- Conversion Operations
- Logic Operations
- Multiplication Operations

- Rotate Operations
- Shift Operations

Add and Subtract Operations

These operations provide addition and/or subtract operations on register and immediate value operands:

- 16-Bit Add or Subtract (AddSub16)
- Vectored 16-Bit Add or Subtract (AddSubVec16)
- 32-bit Add or Subtract (AddSub32)
- Dual 32-bit Add and Subtract (AddSub32Dual)
- 32-Bit Add or Subtract with Carry (AddSubAC0)
- 32-bit Add Constant (AddImm)
- Accumulator Add or Subtract (AddSubAcc)
- Accumulator Add and Extract (AddAccExt)
- Dual Accumulator Add and Subtract to Registers (AddSubAccExt)
- 32-bit Add then Shift (AddSubShift)

16-Bit Add or Subtract (AddSub16)

General Form

Dsp32Alu
DDST0_HL = DREG_L + DREG_L SAT2
DDST0_HL = DREG_L + DREG_H SAT2
DDST0_HL = DREG_H + DREG_L SAT2
DDST0_HL = DREG_H + DREG_H SAT2
DDST0_HL = DREG_L - DREG_L SAT2
DDST0_HL = DREG_L - DREG_H SAT2
DDST0_HL = DREG_H - DREG_L SAT2
DDST0_HL = DREG_H - DREG_H SAT2

Abstract

This instruction adds or subtracts two signed register halves.

See Also (AddSubVec16)

AddSub16 Description

The AddSub16 instruction adds or subtracts two source values and places the result in a destination register with or without result saturation.

AddSub16 accepts any combination of upper and lower half-register operands, and places the results in the upper or lower half of the destination register at the user's discretion.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

In the syntax, where *SAT2* appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSub16 Example

```
/* If r0.l = 0x7000 and r7.l = 0x2000, then . . . */
r4.l = r0.l + r7.l (ns) ;   /* . . . produces r4.l = 0x9000, because no saturation is
enforced */
  /* If r0.l = 0x7000 and r7.h = 0x2000, then . . . */
r4.l = r0.l + r7.h (s) ;
  /* . . . produces r4.l = 0x7FFF, saturated to the maximum positive value */
r0.l = r2.h + r4.l (ns) ;
r1.l = r3.h + r7.h (ns) ;
r4.h = r0.l + r7.l (ns) ;
r4.h = r0.l + r7.h (ns) ;
r0.h = r2.h + r4.l (s) ; /* saturate the result */
r1.h = r3.h + r7.h (ns) ;

r4.l = r0.l - r7.l (ns) ;
r4.l = r0.l - r7.h (s) ; /* saturate the result */
r0.l = r2.h - r4.l (ns) ;
r1.l = r3.h - r7.h (ns) ;
r4.h = r0.l - r7.l (ns) ;
r4.h = r0.l - r7.h (ns) ;
r0.h = r2.h - r4.l (s) ; /* saturate the result */
r1.h = r3.h - r7.h (ns) ;
```

Vectored 16-Bit Add or Subtract (AddSubVec16)

General Form

Dsp32Alu
DREG = DREG AOPL DREG SX
DREG = DREG +|+ DREG , DREG = DREG -|- DREG SXA
DREG = DREG +|- DREG , DREG = DREG -|+ DREG SXA

Abstract

This instruction adds or subtracts two sets of two signed 16-bit vectors, and it deposits them into two destination registers. Optionally, the result of the additions can be saturated. Also, the y inputs can be "crossed" so that instead of adding Rx.h+Ry.h you add Rx.h+Ry.l (and so on). You may also cross the output halves on compute 0.

See Also (AddSub16)

AddSubVec16 Description

The Vector Add / Subtract instruction simultaneously adds and/or subtracts two pairs of registered numbers. It then stores the results of each operation into a separate 32-bit data register or 16-bit half register, according to the syntax used. The destination register for each of the quad or dual versions must be unique.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

The AddSubVec16 instruction supports dual and quad 16-Bit operations.

In the syntax, where *SX* appears (for dual 16-bit operations), substitute a saturation and/or cross output option (s, co, or sco). In the syntax, where *SXA* appears (for quad 16-bit operations), substitute one of the *SX* values, substitute an arithmetic shift right or left option (asr or asl) ASR (arithmetic shift right). The options shown for quad 16-bit operations are scaling options. See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

NOTE: A **special application** of the AddSubVec16 instruction is the FFT butterfly routines in which each of the registers is considered a single complex number often use the Vector Add / Subtract instruction.

```
/* If r1 = 0x0003 0004 and r2 = 0x0001 0002, then . . . */
r0 = r2 +|- r1(co) ;
/* . . . produces r0 = 0xFFFE 0004 */
```

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	**AC1**	**AC0**	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubVec16 Example

```
r5=r3 +|+ r4 ;
   /* dual 16-bit operations, add|add */
r6=r0 -|+ r1(s) ;
   /* same as above, subtract|add with saturation */
r0=r2 +|- r1(co) ;
   /* add|subtract with half-word results crossed over in the destination register */
r7=r3 -|- r6(sco) ;
   /* subtract|subtract with saturation and half-word results crossed over in the
      destination register */
r5=r3 +|+ r4, r7=r3-|-r4 ;
   /* quad 16-bit operations, add|add, subtract|subtract */
r5=r3 +|- r4, r7=r3 -|+ r4 ;
   /* quad 16-bit operations, add|subtract, subtract|add */
r5=r3 +|- r4, r7=r3 -|+ r4(asr) ;
   /* quad 16-bit operations, add|subtract, subtract|add, with all results divided
      by 2 (right shifted 1 place) before storing into destination register */
r5=r3 +|- r4, r7=r3 -|+ r4(asl) ;
   /* quad 16-bit operations, add|subtract, subtract|add, with all results
     multiplied by 2 (left shifted 1 place) before storing into destination register
dual */
```

32-bit Add Constant (AddImm)

General Form

CompI2opD
DREG += imm7

Abstract

This instruction allows the user to add a constant to a register. This instruction does not saturate on overflow

See Also (AddSub32, AddSub32Dual, AddSubAC0)

AddImm Description

The Add Immediate instruction adds a constant value to a register without saturation.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddImm Example

```
r0 += 40 ; /* increment r0 value by 40 and store in r0 */
```

32-bit Add or Subtract (AddSub32)

General Form

Comp3op
DREG = DREG + DREG
DREG = DREG - DREG
Dsp32Alu
DREG = DREG + DREG NSAT
DREG = DREG - DREG NSAT

Abstract

This instruction adds or subtracts two signed registers. The ALU does not saturate the result by default.

See Also (AddSub32Dual, AddSubACO, AddImm)

AddSub32 Description

The AddSub32 instruction adds or subtracts two source values and places the result in a destination register with or without result saturation.

AddSub32 accepts any combination of register operands, and places the results in the destination register at the user's discretion.

This instruction is encoded as a **16-bit instruction** if the *NSAT* option is omitted. The 16-bit encoded instruction 16-bit instruction takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

When the *NSAT* option is included, the instruction is encoded as a **32-bit instruction**. The 32-bit encoded instruction can sometimes save execution time (over a 16-bit encoded instruction), because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

In the syntax, where *NSAT* appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSub32 Example

```
r5 = r2 + r1 ;    /* 16-bit instruction length add, no saturation */
r5 = r2 + r1 (ns) ;    /* same result as above, but 32-bit instruction length */
r5 = r2 + r1 (s) ;    /* saturate the result */

r5 = r2 - r1 ;        /* 16-bit instruction length subtract, no saturation */
r5 = r2 - r1 (ns) ;    /* same result as above, but 32-bit instruction length */
r5 = r2 - r1 (s) ;     /* saturate the result */
```

Dual 32-bit Add and Subtract (AddSub32Dual)

General Form

```
Dsp32Alu
DREG = DREG + DREG , DREG = DREG - DREG  SAT
```

Abstract

This instruction adds and subtracts two signed registers. The ALU does not saturate the result by default.

See Also (`AddSub32`, `AddSubAC0`, `AddImm`)

AddSub32Dual Description

The AddSub32Dual instruction simultaneously adds and subtracts one pair of registered numbers. (The operands of the subtract must be the same as the the operands of the add.) Then, the instruction stores the results of each operation into a separate 32-bit data register with or without result saturation. Each destination register must be unique.

AddSub32Dual accepts any combination of register operands (limited only in that the operands of the subtract and the add must be the same pair of 32-bit data registers), and the instruction places the results in the destination registers at the user's discretion (provided that the destination registers differ from the operand registers).

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

In the syntax, where *SAT* appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	**AC1**	**AC0**	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSub32Dual Example

```
r2=r0+r1, r3=r0-r1 ;   /* dual 32-bit operations */
r2=r0+r1, r3=r0-r1 (s) ;   /* dual 32-bit operations with saturation */
```

32-Bit Add or Subtract with Carry (AddSubAC0)

General Form

Dsp32Alu
DREG = DREG + DREG + ac0 SAT
DREG = DREG - DREG + ac0 - 1 SAT

Abstract

This instruction adds or subtracts two 32-Bit numbers plus a carry bit. This operation is used to implement multi-precision addition and subtraction. Optionally, the user can saturate the result.

See Also (AddSub32, AddSub32Dual, AddImm)

AddSubAC0 Description

The AddSubAC0 instruction adds or subtracts two source values plus a carry bit and places the result in a destination register with or without result saturation.

AddSubAC0 accepts any combination of register operands, and places the results in the destination register at the user's discretion.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

When the *SAT* option is included, the instruction is encoded as a **32-bit instruction**. The 32-bit encoded instruction can sometimes save execution time (over a 16-bit encoded instruction), because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

In the syntax, where *SAT* appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubAC0 Example

```
r5 = r2 + r1 + ac0 ;
/* add with carry, no saturation implied */
r5 = r2 + r1 + ac0 (ns) ;
/* same result as above */
r5 = r2 + r1 + ac0 (s) ;
/* saturate the result */

r5 = r2 - r1 + ac0 - 1 ;
/* sub with carry, no saturation implied */
r5 = r2 - r1 + ac0 -1 (ns) ;
/* same result as above */
r5 = r2 - r1 + ac0 -1 (s) ;
/* saturate the result */
```

The instructions may be used with AddSub32 to implement multi-word addition or subtraction as follows:

```
/* 64-bit addition: R1:0=R3:2+R5:4 */
R0 = R2 + R4 ;
R1 = R3 + R5 + AC0 ;
/* 64-bit subtraction: R1:0=R3:2-R5:4*/
R0 = R2 - R4 ;
R1 = R3 - R5 + AC0 - 1 ;
```

Accumulator Add and Extract (AddAccExt)

General Form

Dsp32Alu
DREG = (a0 += a1)
DDST0_HL = (a0 += a1)

Abstract

This instruction adds the two signed accumulators together, then extracts the result to a register.

See Also (AddSubAcc, AddSubAccExt)

AddAccExt Description

The AddAccExt instruction increments the 40-bit A0 accumulator register by A1 with saturation at 40 bits, then extracts the result into a 32-bit register with saturation at 32 bits, or a 16 bit register with saturation at 16 bits.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	**AV0S**	**AV0**
...	...	AC1	**AC0**	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddAccExt Example

```
r5 = (a0 += a1) ;
r2.l = (a0 += a1) ;
r5.h = (a0 += a1) ;
```

Accumulator Add or Subtract (AddSubAcc)

General Form

Dsp32Alu
a0 += a1
a0 += a1 (w32)
a0 -= a1
a0 -= a1 (w32)

Abstract

This instruction adds or subtracts two signed accumulators. The ALU saturates the result on overflow.

See Also (AddAccExt, AddSubAccExt)

AddSubAcc Description

The AddSubAcc instruction adds or subtracts two source values in the accumulator registers and places the result in a destination acculator register with or without result saturation.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

The syntax of this instruction provides optional saturation/sign-extension of the result.

- (W32) - signed saturate the result at 32 bits, sign extended

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	**AV0S**	**AV0**
...	...	AC1	**AC0**	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubAcc Example

```
a0 += a1 ; /* no saturation */
a0 += a1 (w32) ; /* signed saturate at 32 bits, sign extended */

a0 -= a1 ; /* no saturation */
a0 -= a1 (w32) ; /* signed saturate at 32 bits, sign extended */
```

Dual Accumulator Add and Subtract to Registers (AddSubAccExt)

General Form

Dsp32Alu
DREG = a1 + a0, DREG = a1 - a0 SAT
DREG = a0 + a1, DREG = a0 - a1 SAT

Abstract

This instruction adds and subtracts the two accumulators together, then extracts the results.

See Also (AddSubAcc, AddAccExt)

AddSubAccExt Description

The AddSubAccExt instruction simultaneously adds and subtracts the two 40-bit accumulator registers. Then, the instruction stores the results of each operation into a separate 32-bit data register with or without result saturation. Each destination register must be unique.

AddSubAccExt accepts any combination of register operands, and places the results in the destination registers at the user's discretion.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

In the syntax, where *SAT* appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	**AC1**	**AC0**	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubAccExt Example

```
r4=a1+a0, r6=a1-a0 ;
   /* dual 40-bit accumulator operations with no saturation, A0 added/subtracted from
A1 */
r4=a0+a1, r6=a0-a1(s) ;
   /* dual 40-bit accumulator operations with saturation, A1 subtracted from A0 */
```

32-bit Add then Shift (AddSubShift)

General Form

ALU2op
DREG =(DREG + DREG)<< 1
DREG =(DREG + DREG)<< 2

Abstract

This instruction adds then shifts left one or two places. This instruction always saturates on overflow.

AddSubShift Description

The AddSubShift instruction combines an addition operation with a one- or two-place logical shift left. The left shift accomplishes a x2 (for shift 1) or x4 (for shift 2) multiplication on sign-extended numbers. The instruction always saturates on overflow.

The first input operand register must be the same as the destination register.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubShift Example

```
r3 = (r3+r2)<<1 ;     /* r3 = (r3 + r2) * 2 */
r3 = (r3+r2)<<2 ;     /* r3 = (r3 + r2) * 4 */
```

Bit Operations

These operations provide bitwise shift operations on registers operands:

- Ones Count (Shift_Ones)
- Redundant Sign Bits (Shift_SignBits32)
- Redundant Sign Bits (Shift_SignBitsAcc)
- Bit Mux (BitMux)
- Bit Modify (Shift_BitMod)
- Bit Test (Shift_BitTst)
- Deposit Bits (Shift_Deposit)
- Extract Bits (Shift_Extract)

Ones Count (Shift_Ones)

General Form

Dsp32Shf
DREG_L = ones DREG

Abstract

This instruction counts the number of 1's in a XOP register.

See Also (Shift_SignBits32, Shift_SignBitsAcc)

Shift_Ones Description

The Shift_Ones instruction (one's-population count) loads the number of 1's contained in the souce register (*DSRC1*) into the lower half of the destination register (*DDST_L*).

The range of possible values loaded into *DDST_L* is 0 through 32.

The *DDST_L* and *DSRC1* can be the same D-register. Otherwise, the One's-Population Count instruction does not modify the contents of *DSRC1*.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

Shift_Ones Example

```
r3.l = ones r7 ;
```

If R7 contains 0xA5A5A5A5, R3.L contains the value 16, or 0x0010.

If R7 contains 0x00000081, R3.L contains the value 2, or 0x0002.

Redundant Sign Bits (Shift_SignBits32)

General Form

Dsp32Shf
DREG_L = signbits DREG
DREG_L = signbits DREG_L
DREG_L = signbits DREG_H

Abstract

This instruction returns the number of redundant sign bits. For example, if there are five sign bits, this instruction returns 4. The result can then be used with ASHIFT to normalize the data.

See Also (Shift_Ones, Shift_SignBitsAcc)

Shift_SignBits32 Description

The SignBits32 instruction returns the number of sign bits in a number, and can be used in conjunction with a shift to normalize numbers. This instruction can operate on 16-bit or 32-bit input numbers.

- For a 16-bit input, Sign Bit returns the number of leading sign bits minus one, which is in the range 0 through 15. There are no special cases. An input of all zeros returns +15 (all sign bits), and an input of all ones also returns +15.
- For a 32-bit input, Sign Bit returns the number of leading sign bits minus one, which is in the range 0 through 31. An input of all zeros or all ones returns +31 (all sign bits).

The result of the SignBits32 instruction can be used directly as the argument to an arithmetic shift instruction (AShift) to normalize the number. Resultant numbers will be in the following formats (S == signbit, M == magnitude bit).

16-bit:	S.MMM MMMM MMMM MMMM
32-bit:	S.MMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM

In addition, the SignBits32 instruction result can be subtracted directly to form the new exponent.

The SignBits32 instruction does not implicitly modify the input value. For 32-bit and 16-bit input, the destination register (*DDST_L*) and source sample register (*DSRC1*) can be the same D-register. Doing this explicitly modifies the *DSRC1*.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

Shift_SignBits32 Example

```
r2.l = signbits r7 ;
r1.l = signbits r5.l ;
r0.l = signbits r4.h ;
```

Redundant Sign Bits (Shift_SignBitsAcc)

General Form

Dsp32Shf
DREG_L = signbits a0
DREG_L = signbits a1

Abstract

This instruction returns the number of redundant sign bits. For example, if there are five sign bits, this instruction returns 4. The result can then be used with ASHIFT to normalize the data.

See Also (Shift_Ones, Shift_SignBits32)

Shift_SignBitsAcc Description

The SignBitsAcc instruction returns the number of sign bits in a number, and can be used in conjunction with a shift to normalize numbers. This instruction can operate on 40-bit input numbers.

- For a 40-bit Accumulator input, Sign Bit returns the number of leading sign bits minus 9, which is in the range -8 through +31. A negative number is returned when the result in the Accumulator has expanded into the extension bits; the corresponding normalization will shift the result down to a 32-bit quantity (losing precision). An input of all zeros or all ones returns +31.

The result of the SignBitsAcc instruction can be used directly as the argument to an arithmetic shift instruction (AShift) to normalize the number. Resultant numbers will be in the following formats (S == signbit, M == magnitude bit).

40-bit:	SSSS SSSS S.MMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM

In addition, the SignBitsAcc instruction result can be subtracted directly to form the new exponent.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

Shift_SignBitsAcc Example

```
r6.l = signbits a0 ;
r5.l = signbits a1 ;
```

Bit Mux (BitMux)

General Form

Dsp32Shf
bitmux (DREG , DREG , a0) (asr)
bitmux (DREG , DREG , a0) (asl)

Abstract

This instruction merges two bit streams into the accumulator. Each time you call this instruction, it takes a single bit from the two source registers, muxes them together, and deposits them into the accumulator. The streams can be taken from the MSBs of the register pair and shifted into the LSBs of the accumulator (ASL) or taken from the LSBs of the registers and deposited into the MSBs of the accumulator (ASR).

See Also (Shift_BitMod, Shift_BitTst)

BitMux Description

The BitMux instruction merges bit streams.

The instruction has two versions, shift right and shift left. This instruction overwrites the contents of source 1 (*DSRC1*) and source 0 (*DSRC0*). See the **Contents Before Shift** table, **A Shift Right Instruction** table, and **A Shift Left Instruction** table.

In the Shift Right version, the processor performs the following sequence.

1. Right shift Accumulator A0 by one bit. Right shift the LSB of *DSRC1* into the MSB of the Accumulator.
2. Right shift Accumulator A0 by one bit. Right shift the LSB of *DSRC0* into the MSB of the Accumulator.

In the Shift Left version, the processor performs the following sequence.

1. Left shift Accumulator A0 by one bit. Left shift the MSB of *DSRC0* into the LSB of the Accumulator.
2. Left shift Accumulator A0 by one bit. Left shift the MSB of *DSRC1* into the LSB of the Accumulator.

DSRC1 and *DSRC0* must not be the same D-register.

Table 8-1: Contents Before Shift

IF	39.....32	31.....24	23.....16	15.....8	7.....0
source_1:		xxxx xxxx	xxxx xxxx	xxxx xxxx	xxxx xxxx
source_0:		yyyy yyyy	yyyy yyyy	yyyy yyyy	yyyy yyyy
Accumulator A0:	zzzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzzz

Table 8-2: A Shift Right Instruction

IF	39.....32	31.....24	23.....16	15.....8	7.....0
source_1:[1]		0xxx xxxx	xxxx xxxx	xxxx xxxx	xxxx xxxx
source_0:[2]		0yyy yyyy	yyyy yyyy	yyyy yyyy	yyyy yyyy
Accumulator A0:[3]	yxzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzzz

1. source_1 is shifted right 1 place
2. source_0 is shifted right 1 place
3. Accumulator A0 is shifted right 2 places

Table 8-3: A Shift Left Instruction

IF	39.....32	31.....24	23.....16	15.....8	7.....0
source_1:[1]		xxxx xxxx	xxxx xxxx	xxxx xxxx	xxxx xxx0
source_0:[2]		yyyy yyyy	yyyy yyyy	yyyy yyyy	yyyy yyy0
Accumulator A0:[3]	zzzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzzz	zzzz zzyx

1. source_1 is shifted left 1 place
2. source_0 is shifted left 1 place
3. Accumulator A0 is shifted left 2 places

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

BitMux Example

```
bitmux (r2, r3, a0) (asr);   /* right shift*/
```

- If
  - R2=0b1010 0101 1010 0101 1100 0011 1010 1010
  - R3=0b1100 0011 1010 1010 1010 0101 1010 0101
  - A0=0b0000 0000 0000 0000 0000 0000 0000 0000 0000 0111

  then the Shift Right instruction produces:
  - R2=0b0101 0010 1101 0010 1110 0001 1101 0101
  - R3=0b0110 0001 1101 0101 0101 0010 1101 0010
  - A0=0b1000 0000 0000 0000 0000 0000 0000 0000 0000 0001

```
bitmux (r3, r2, a0) (asl);   /* left shift*/
```

- If
  - R3=0b1010 0101 1010 0101 1100 0011 1010 1010
  - R2=0b1100 0011 1010 1010 1010 0101 1010 0101
  - A0=0b0000 0000 0000 0000 0000 0000 0000 0000 0000 0111

  then the Shift Left instruction produces:
  - R2=0b0100 1011 0100 1011 1000 0111 0101 0100
  - R3=0b1000 0111 0101 0101 0100 1011 0100 1010
  - A0=0b0000 0000 0000 0000 0000 0000 0000 0000 0001 1111

Bit Modify (Shift_BitMod)

General Form

Logi2Op
bitset (DREG , uimm5)
bittgl (DREG , uimm5)
bitclr (DREG , uimm5)

Abstract

This instruction takes the data register specified and clears, sets, or toggles a bit.

See Also (BitMux, Shift_BitTst)

Shift_BitMod Description

The BitMod instruction includes BitSet, BitTgl, and BitTst forms:

- The BitSet (bit set) instruction sets the bit designated by the bit position (source immediate value, *SRCI*) in the specified D-register destination (*DDST*). It does not affect other bits in the D-register.

  The *SRCI* range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

- The BitTgl (bit toggle) instruction inverts the bit designated by *SRCI* in the specified D-register. The instruction does not affect other bits in the D-register.

  The *SRCI* range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

- The BitClr (bit clear) instruction clears the bit designated by *SRCI* in the specified D-register. It does not affect other bits in that register.

  The *SRCI* range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

This **16-bit instruction** instruction takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift_BitMod Example

BitSet Example

```
bitset (r2, 7) ;   /* set bit 7 (the eighth bit from LSB) in R2 */
```

For example, if `R2` contains 0x00000000 before this instruction, it contains 0x00000080 after the instruction.

BitTgl Example

```
bittgl (r2, 24) ;   /* toggle bit 24 (the 25th bit from LSB in R2 */
```

For example, if `R2` contains 0xF1FFFFFF before this instruction, it contains 0xF0FFFFFF after the instruction. Executing the instruction a second time causes the register to contain 0xF1FFFFFF.

BitClr Example

```
bitclr (r2, 3) ;   /* clear bit 3 (the fourth bit from LSB) in R2 */
```

For example, if `R2` contains 0xFFFFFFFF before this instruction, it contains 0xFFFFFFF7 after the instruction.

Bit Test (Shift_BitTst)

General Form

Logi2Op
cc = !bittst (DREG , uimm5)
cc = bittst (DREG , uimm5)

Abstract

This instruction sets CC bits if the specified condition is true. In the bittst case, the CC bit is set if the specified bit is a 1. For the !bittst case, it is set if the bit is a zero.

See Also (BitMux, Shift_BitMod)

Shift_BitTst Description

The Bit Test instruction sets or clears the CC bit, based on the bit designated by the bit position (source immediate value, *SRCI*) in the specified D-register destination.

One version tests whether the specified bit is set; the other tests whether the bit is clear. The instruction does not affect other bits in the D-register.

The *SRCI* range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

This **16-bit instruction** instruction takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	**CC**	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift_BitTst Example

```
cc = bittst (r7, 15) ;   /* test bit 15 TRUE in R7 */
```

For example, if R7 contains 0xFFFFFFFF before this instruction, CC is set to 1, and R7 still contains 0xFFFFFFFF after the instruction.

```
cc = ! bittst (r3, 0) ;   /* test bit 0 FALSE in R3 */
```

If R3 contains 0xFFFFFFFF, this instruction clears CC to 0.

Deposit Bits (Shift_Deposit)

General Form

Dsp32Shf
DREG = deposit (DREG , DREG)
DREG = deposit (DREG , DREG) (x)

Abstract

The bit field deposit instruction merges the background data in the first source register with a foreground bit field in the second source register and saves the result into the destination register.

See Also (Shift_Extract)

Shift_Deposit Description

The bit field deposit instruction merges the background data in the first source register (DSRC1) with a foreground bit field in the second source register (DSRC0). The foreground bit field is specified with the field length, a value in the range 0 to 16 in DSRC0.b0, the field position, a value in the range 0 to 31 in DSRC0.b1, and the field value, a 16-bit value in DSRC0.h. The result value is constructed by truncating the field value to field length and depositing at field position in the background value. By default the bits in the result above field position + field length are taken from the background value. The (X) syntax sets these bits to the most significant bit of the truncated field value. The result value is stored in the destination register (DDST).

```
Field deposit
       31              16 15     8 7      0
      +-----------------+--------+--------+
src0: |xxxxxxxxxxxxxSNN |    P   |    L   |   L--length, P--position
      +-----------------+--------+--------+
      +-----------------+-----------------+
src1  |bbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbb|
      +-----------------+-----------------+

      +-----------------+-----------------+
dst0: |bbbbbbbbbbbbbbSNN bbbbbbbbbbbbbbbb|  SN--inserted field in src0
      +-----------------+-----------------+   b--previous contents of src1

with sign extension
      +-----------------+-----------------+
dst0: |SSSSSSSSSSSSSSSNN bbbbbbbbbbbbbbbb|
      +-----------------+-----------------+
```

The operation writes the foreground bit field of length *L* over the background bit field with the foreground LSB located at bit *p* of the background.

There are a number of *boundary cases* related to Shift_Deposit instruction operation that should be considered.

- Unsigned syntax, L = 0: The architecture copies *DSRC1* contents without modification into *DDST*. By definition, a foreground of zero length is transparent.
- Sign-extended, L = 0 and p = 0: This case loads 0x0000 0000 into *DDST*. The sign of a zero length, zero position foreground is zero; therefore, sign-extended is all zeros.
- Sign-extended, L = zero and p = non-zero: The instruction copies the lower order bits of *DSRC1* below position *p* into *DDST*, then sign-extends that number. The foreground value has no effect. For instance, if:
  - *DSRC1* = 0x0000 8123,
  - L = 0, and
  - p = 16,
  - then:
  - *DDST* = 0xFFFF 8123.

  In this example, the architecture copies bits 15-0 from *DSRC1* into *DDST*, then sign-extends that number.
- Both Syntaxes, L > 16: Treats L as 16
- Both Syntaxes, (L + p) > 32: Any foreground bits that fall outside the range 31-0 are truncated.

The Bit Field Deposit instruction does not modify the contents of the two source registers. One of the source registers can also serve as *DDST*.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift_Deposit Example

Bit Field Deposit Unsigned

```
r7 = deposit (r4, r3) ;
```

- If
  - R4=0b1111 1111 1111 1111 1111 1111 1111 1111 where this is the background bit field
  - R3=0b0000 0000 0000 0000 0000 0111 0000 0011 where bits 31-16 are the foreground bit field, bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Deposit (unsigned) instruction produces:
  - R7=0b1111 1111 1111 1111 1111 1100 0111 1111
- If
  - R4=0b1111 1111 1111 1111 1111 1111 1111 1111 where this is the background bit field
  - R3=0b0000 0000 1111 1010 0000 1101 0000 1001 where bits 31-16 are the foreground bit field, bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Deposit (unsigned) instruction produces:
  - R7=0b1111 1111 1101 1111 0101 1111 1111 1111

Bit Field Deposit Sign-Extended

```
r7 = deposit (r4, r3) (x) ;   /* sign-extended*/
```

- If
  - R4=0b1111 1111 1111 1111 1111 1111 1111 1111 where this is the background bit field
  - R3=0b0101 1010 0101 1010 0000 0111 0000 0011 where bits 31-16 are the foreground bit field, bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Deposit (unsigned) instruction produces:
  - R7=0b0000 0000 0000 0000 0000 0001 0111 1111
- If
  - R4=0b1111 1111 1111 1111 1111 1111 1111 1111 where this is the background bit field
  - R3=0b0000 1001 1010 1100 0000 1101 0000 1001 where bits 31-16 are the foreground bit field, bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Deposit (unsigned) instruction produces:
  - R7=0b1111 1111 1111 0101 1001 1111 1111 1111

Extract Bits (Shift_Extract)

General Form

Dsp32Shf
DREG = extract (DREG , DREG_L) (z)
DREG = extract (DREG , DREG_L) (x)

Abstract

This instruction extracts specified bits from the first source register and sign or zero extends the bits to 32-bits and writes the bits to the destination register.

See Also (Shift_Deposit)

Shift_Extract Description

Extracts specified bits from the first source register (DSRC1) and writes them to the low order bits of the destination register. The bits are specified with the second source register (DSCR0). The bit position is stored in DSRC0.b1 and the length is stored in DSCR0.b0.. The field is either sign extended or zero extended to fill the 32-bit output register. ((Z) zero fills, (X) sign extends))

```
Field extraction
      31     16 15 8 7   0
      +--------+----+----+
src0: |xxxxxxxx|  P |  L |   L--length, P--position
      +--------+----+----+
      +--------+---------+
src1  |bbbbbbbbbbSNNbbbbb|
      +--------+---------+

      +--------+---------+
dst0: |0000000000000000SNN|  SN--inserted field in hi half of src0
      +--------+---------+   b--previous contents of src1
x--unused
with sign extension          0--zero
      +--------+---------+
dst0: |SSSSSSSSSSSSSSSSSNN|
      +--------+---------+
```

The operation reads the pattern bit field of length L from the scene bit field, with the pattern LSB located at bit p of the scene. See "Example", below, for more.

There are a number of *boundary cases* related to Shift_Extract instruction operation that should be considered.

If (p + L) > 32: In the zero-extended and sign-extended versions of the instruction, the architecture assumes that all bits to the left of the *DSRC1* are zero. In such a case, the user is trying to access more bits than the register actually contains. Consequently, the architecture fills any undefined bits beyond the MSB of the *DSRC1* with zeros.

The Bit Field Extraction instruction does not modify the contents of the two source registers. One of the source registers can also serve as *DDST*.

The user has the choice of using the `(X)` option syntax to perform sign-extend extraction or the `(Z)` option syntax to perform zero-extend extraction.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift_Extract Example

Bit Field Extraction Unsigned

```
r7 = extract (r4, r3.l) (z) ;   /* zero-extended*/
```

- If
  - `R4=0b1010 0101 1010 0101 1100 0011 1010 1010` where this is the scene bit field
  - `R3=0bxxxx xxxx xxxx xxxx 0000 0111 0000 0100` where bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Extraction (unsigned) instruction produces:
  - `R7=0b0000 0000 0000 0000 0000 0000 0000 0111`
- If
  - `R4=0b1010 0101 1010 0101 1100 0011 1010 1010` where this is the scene bit field

– R3=0bxxxx xxxx xxxx xxxx 0000 1101 0000 1001 where bits 15-8 are the position, and bits 7-0 are the length

then the Bit Field Extraction (unsigned) instruction produces:

– R7=0b0000 0000 0000 0000 0000 0001 0010 1110

Bit Field Extraction Sign-Extended

```
r7 = extract (r4, r3.l) (x) ;   /* sign-extended*/
```

- If
  - R4=0b1010 0101 1010 0101 1100 0011 1010 1010 where this is the scene bit field
  - R3=0bxxxx xxxx xxxx xxxx 0000 0111 0000 0100 where bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Extraction (sign-extended) instruction produces:

  – R7=0b0000 0000 0000 0000 0000 0000 0000 0111
- If
  - R4=0b1010 0101 1010 0101 1100 0011 1010 1010 where this is the scene bit field
  - R3=0bxxxx xxxx xxxx xxxx 0000 1101 0000 1001 where bits 15-8 are the position, and bits 7-0 are the length

  Then the Bit Field Extraction (sign-extended) instruction produces:

  – R7=0b1111 1111 1111 1111 1111 1111 0010 1110

Comparison Operations

These operations provide 16- and 32-bit maximum/minimum comparison and array search operations on register operands:

- Vectored 16-Bit Maximum (Max16Vec)
- Vectored 16-Bit Minimum (Min16Vec)
- 32-bit Maximum (Max32)
- 32-Bit Minimum (Min32)
- Vectored 16-Bit Search (Search)

A number of other comparison operations using the condition code (CC) bit are available:

- 32-Bit Pointer Register Compare and Set CC (CCFlagP)
- Accumulator Compare and Set CC (CompAccumulators)
- 32-Bit Register Compare and Set CC (CompRegisters)

Vectored 16-Bit Maximum (Max16Vec)

General Form

Dsp32Alu
DREG = max(DREG , DREG) (v)

Abstract

This instruction calculates the maximum of two pairs of signed 16-Bit words.

See Also (Min16Vec)

Max16Vec Description

The vector maximum instruction returns the maximum value (meaning the largest positive value, nearest to 0x7FFF) of the 16-bit half-word source registers to the destination register.

The instruction compares the upper half-words of the first source register and the second source register, then the instruction returns that maximum to the upper half-word of the destination register. The instruction also compares the lower half-words of the first source register and the second source register, then the instruction returns the maximum to the lower half-word of the destination register. The result is a concatenation of the two 16-bit maximum values.

The vector maximum instruction does not implicitly modify input values. The destination register can be the same D-register as one of the source registers. Doing this explicitly modifies that source register.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Max16Vec Example

```
r7 = max (r1, r0) (v) ;
```

- Assume R1 = 0x0007 0000 and R0 = 0x0000 000F, then R7 = 0x0007 000F.
- Assume R1 = 0xFFF7 8000 and R0 = 0x000A 7FFF, then R7 = 0x000A 7FFF.
- Assume R1 = 0x1234 5678 and R0 = 0x0000 000F, then R7 = 0x1234 5678.

Vectored 16-Bit Minimum (Min16Vec)

General Form

Dsp32Alu
DREG = min(DREG , DREG) (v)

Abstract

This instruction calculates the minimum of two pairs of signed 16-bit words.

See Also (Max16Vec)

Min16Vec Description

The Vector Minimum instruction returns the minimum value (the most negative value or the value closest to 0x8000) of the 16-bit half-word source registers to the destination register.

This instruction compares the upper half-words of the source registers and returns that minimum to the upper half-word of the destination register. It also compares the lower half-words of the source registers and returns that minimum to the lower half-word of the destination register. The result is a concatenation of the two 16-bit minimum values.

The input values are not implicitly modified by this instruction. The destination register can be the same D-register as one of the source registers. Doing this explicitly modifies that source register.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Min16Vec Example

```
r7 = min (r1, r0) (v) ;
```

- Assume R1 = 0x0007 0000 and R0 = 0x0000 000F, then R7 = 0x0000 0000.
- Assume R1 = 0xFFF7 8000 and R0 = 0x000A 7FFF, then R7 = 0xFFF7 8000.
- Assume R1 = 0x1234 5678 and R0 = 0x0000 000F, then R7 = 0x0000 000F.

32-bit Maximum (Max32)

General Form

Dsp32Alu
DREG = max(DREG , DREG)

Abstract

This instruction calculates the maximum of two signed 32-bit values.

See Also (Min32)

Max32 Description

The maximum instruction returns the maximum, or most positive, value of the source registers.

The maximum instruction does not implicitly modify input values. The destination register can be the same D-register as one of the source registers. Doing this explicitly modifies the source register.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Max32 Example

```
r5 = max (r2, r3) ;
```

- Assume R2 = 0x00000000 and R3 = 0x0000000F, then R5 = 0x0000000F.
- Assume R2 = 0x80000000 and R3 = 0x0000000F, then R5 = 0x0000000F.

- Assume R2 = 0xFFFFFFFF and R3 = 0x0000000F, then R5 = 0x0000000F.
- Assume R2 = 0x00000010 and R3 = 0x0000000F, then R5 = 0x00000010.

32-Bit Minimum (Min32)

General Form

Dsp32Alu
DREG = min(DREG , DREG)

Abstract

This instruction calculates the minimum of two signed 32-bit values.

See Also (Max32)

Min32 Description

The minimum instruction returns the minimum value of the source registers to the dest_reg. (The minimum value of the source registers is the value closest to −∞.)

The minimum instruction does not implicitly modify input values. The destination register can be the same D-register as one of the source registers. Doing this explicitly modifies the source register.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Min32 Example

```
r5 = min (r2, r3) ;
```

- Assume R2 = 0x00000000 and R3 = 0x0000000F, then R5 = 0x00000000.
- Assume R2 = 0x80000000 and R3 = 0x0000000F, then R5 = 0x80000000.

- Assume R2 = 0xFFFFFFFF and R3 = 0x0000000F, then R5 = 0xFFFFFFFF.
- Assume R2 = 0x00000010 and R3 = 0x0000000F, then R5 = 0x0000000F.

Vectored 16-Bit Search (Search)

General Form

Dsp32Alu
(DREG , DREG) = search DREG (gt)
(DREG , DREG) = search DREG (ge)
(DREG , DREG) = search DREG (lt)
(DREG , DREG) = search DREG (le)

Abstract

This instruction is used in a loop to locate a minimum or maximum in an array. For each compute unit, a value is compared against the current signed max or min in the accumulator. Two values are tested at a time, the current winner will be stored in the accumulator and the current value of P0 will be written to the result register if the comparison is true.

Search Description

This instruction is used in a loop to locate a maximum or minimum element in an array of 16-bit packed data. Two values are tested at a time. The vector search instruction compares two 16-bit, signed half-words to values stored in the Accumulators. Then, it conditionally updates each accumulator and destination pointer based on the comparison. Pointer register P0 is always the implied array pointer for the elements being searched.

More specifically, the signed high half-word of *src_reg* is compared in magnitude with the 16 low-order bits in A1. If *src_reg_hi* meets the comparison criterion, then A1 is updated with *src_reg_hi*, and the value in pointer register P0 is stored in *dest_pointer_hi*. The same operation is performed for *src_reg_low* and A0.

Based on the search mode specified in the syntax, the instruction tests for maximum or minimum signed values.

Values are sign extended when copied into the accumulator(s). See the examples for one way to implement the search loop. After the vector search loop concludes, A1 and A0 hold the two surviving elements, and *dest_pointer_hi* and *dest_pointer_lo* contain their respective addresses. The next step is to select the final value from these two surviving elements.

Modes

The four supported compare modes are specified by the mandatory searchmode flag.

Table 8-4: Compare Modes

Table 8-5:

Mode	Description
(GT)	Greater than. Find the location of the first maximum number in an array.
(GE)	Greater than or equal. Find the location of the last maximum number in an array.
(LT)	Less than. Find the location of the first minimum number in an array.
(LE)	Less than or equal. Find the location of the last minimum number in an array.

Summary (assumed Pointer P0)

src_reg_hi

Compared to least significant 16 bits of A1. If compare condition is met, overwrites lower 16 bits of A1 and copies P0 into dest_pointer_hi.

src_reg_lo

Compared to least significant 16 bits of A0. If compare condition is met, overwrites lower 16 bits of A0 and copies P0 into dest_pointer_lo.

This **32-bit instruction** can be issued in parallel with the combination of one 16-bit length load instruction to the P0 register and one 16-bit NOP. No other instructions can be issued in parallel with the vector search instruction. Note the following legal and illegal forms.

```
(r1, r0) = search r2 (LT) || r2 = [p0++p3]; /* ILLEGAL */
(r1, r0) = search r2 (LT) || r2 = [p0++]; /* LEGAL */
(r1, r0) = search r2 (LT) || r2 = [p0++]; /* LEGAL */
```

Search Example

```
/* Initialize Accumulators with appropriate value for the type of search. */
  r0.l=0x7fff ;
  r0.h=0 ;
  a0=r0 ; /* max positive 16-bit value */
  a1=r0 ; /* max positive 16-bit value */
/* Initialize R2. */
  r2=[p0++] ;
/* Assume P1 is initialized to the size of the vector length. */
  LSETUP (loop_, loop_) LC0=P1>>1 ; /* set up the loop */
  loop_: (r1,r0) = SEARCH R2 (LE) || R2=[P0++];
    /* search for the last minimum in all but the
    last element of the array */
    (r1,r0) = SEARCH R2 (LE);
    /* finally, search the last element */
/* The lower 16 bits of A1 and A0 contain the last minimums of the
array. R1 contains the value of P0 corresponding to the value in
A1. R0 contains the value of P0 corresponding to the value in A0.
Next, compare A1 and A0 together and R1 and R0 together to find
the single, last minimum in the array.
```

```
Note: In this example, the resulting pointers are past the actual
surviving array element due to the post-increment operation. */
  cc = a0 <= a1 ;
  r0 += -4 ;
  r1 += -2 ;
  if !cc r0 = r1 ;
/* the pointer to the survivor is in r0 */
```

Conversion Operations

These operations provide absolute value, negate, pass, and saturate operations on register operands:

- Vectored 16-Bit Absolute Value (Abs2x16)
- 32-bit Absolute Value (Abs32)
- Accumulator0 Absolute Value (AbsAcc0)
- Accumulator Absolute Value (AbsAcc1)
- Accumulator Absolute Value (AbsAccDual)
- Vectored 16-bit Negate (Neg16Vec)
- 32-Bit Negate (Neg32)
- Accumulator0 Negate (NegAcc0)
- Accumulator1 Negate (NegAcc1)
- Dual Accumulator Negate (NegAccDual)
- Fractional 32-bit to 16-Bit Conversion (Pass32Rnd16)
- Accumulator0 32-Bit Saturate (ALU_SatAcc0)
- Accumulator1 32-Bit Saturate (ALU_SatAcc1)
- Dual Accumulator 32-Bit Saturate (ALU_SatAccDual)

Vectored 16-Bit Absolute Value (Abs2x16)

General Form

Dsp32Alu
DREG = abs DREG (v)

Abstract

This instruction calculates the absolute values of a pair of signed 16-bit words. Saturation only applies when the input is 0x8000.

Abs2x16 Description

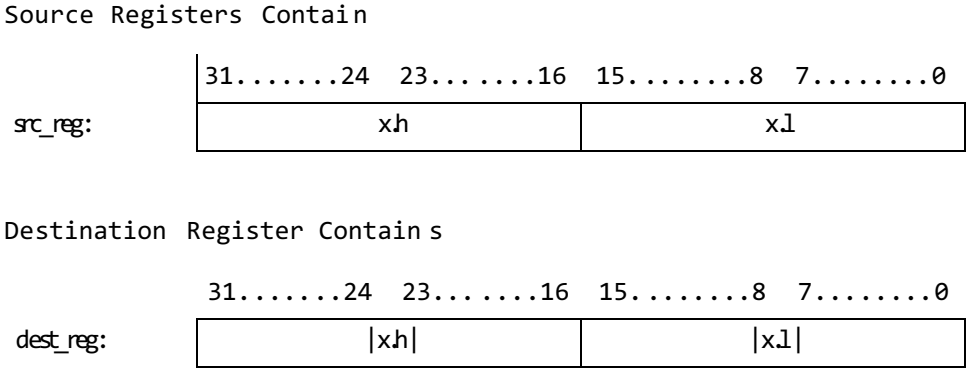
The vector absolute value instruction calculates the individual absolute values of the upper and lower halves of a single 32-bit data register. The results are placed into a 32-bit *dest_reg*, using the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination.

This instruction saturates the result.

For example, as shown in the figure, if the source register contains the data shown, the destination register receives the data shown.

Figure 8-3: Source/Destination Value Placement



This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Abs2x16 Example

```
/* If r1 = 0xFFFF 7FFF, then . . . */
r3 = abs r1 (v) ;
/* . . . produces 0x0001 7FFF */
```

32-bit Absolute Value (Abs32)

General Form

Dsp32Alu
DREG = abs DREG

Abstract

This instruction calculates the absolute value of the 32-bit input. Saturation only applies when the input is 0x80000000.

Abs32 Description

This instruction calculates the absolute value of a 32-bit data register and stores it into another 32-bit data register. Calculation is done according to the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination. Saturation is automatically performed with the instruction, so taking the absolute value of the largest- magnitude negative number returns the largest-magnitude positive number.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Abs32 Example

```
r3 = abs r1 ;
```

Accumulator0 Absolute Value (AbsAcc0)

General Form

Dsp32Alu
a0 = abs a0
a0 = abs a1

Abstract

This instruction calculates the absolute value of the A0 (accumulator 0) register.

See Also (AbsAcc1, AbsAccDual)

AbsAcc0 Description

This instruction takes the absolute value of a 40-bit input value in an accumulator and produces a 40-bit result. Calculation is done according to the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination. Saturation is automatically performed with the instruction, so taking the absolute value of the largest- magnitude negative number returns the largest-magnitude positive number.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AbsAcc0 Example

```
a0 = abs a0 ;
a0 = abs a1 ;
```

Accumulator Absolute Value (AbsAcc1)

General Form

Dsp32Alu
a1 = abs a0
a1 = abs a1

Abstract

This instruction calculates the absolute value of the A1 (accumulator 1) register.

See Also (AbsAcc0, AbsAccDual)

AbsAcc1 Description

This instruction takes the absolute value of a 40-bit input value in a register and produces a 40-bit result. Calculation is done according to the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination. Saturation is automatically performed with the instruction, so taking the absolute value of the largest- magnitude negative number returns the largest-magnitude positive number.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	**AV1S**	**AV1**	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AbsAcc1 Example

```
a1 = abs a0 ;
a1 = abs a1 ;
```

Accumulator Absolute Value (AbsAccDual)

General Form

Dsp32Alu
a1 = abs a1, a0 = abs a0

Abstract

This instruction calculates the absolute value of the A0 (accumulator 0) register and A1 (accumulator 1) register.

See Also (AbsAcc0, AbsAcc1)

AbsAccDual Description

This instruction performs the ABS operation on both accumulators by a single instruction, taking the absolute value of 40-bit input values in two registers and producing two 40-bit results. Calculation is done according to the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination. Saturation is automatically performed with the instruction, so taking the absolute value of the largest- magnitude negative number returns the largest-magnitude positive number.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AbsAccDual Example

```
a1 = abs a1, a0=abs a0 ;
```

Vectored 16-bit Negate (Neg16Vec)

General Form

Dsp32Alu
DREG =- DREG (v)

Abstract

This instruction negates a pair of 16-bit words. The maximum negative inputs (0x8000) saturates to maximum positive.

Neg16Vec Description

The vector negate instruction returns the same magnitude with the opposite arithmetic sign, saturated for each 16-bit half-word in the source. The instruction calculates by subtracting the source from zero.

For more information, see the Saturation section in the Introduction chapter.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	**AC1**	**AC0**	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Neg16Vec Example

```
r5 =-r3 (v) ;
/* R5.H becomes the negative of R3.H and R5.L becomes the negative of R3.L.  */
/* If r3 = 0x0004 7FFF the result is r5 = 0xFFFC 8001 */
```

32-Bit Negate (Neg32)

General Form

ALU2op
DREG =- DREG
Dsp32Alu
DREG =- DREG NSAT

Abstract

This instruction negates the 32-bit input. The maximum negative inputs (0x80000000) saturates to maximum positive, if saturation is specified..

Neg32 Description

The negate (two's-complement) instruction returns the same magnitude with the opposite arithmetic sign. The instruction calculates by subtracting from zero.

The 32-bit version of the negate (two's-complement) instruction is offered with or without saturation. The only case where the nonsaturating negate would overflow is when the input value is 0x8000 0000. The saturating version returns 0x7FFF FFFF; the nonsaturating version returns 0x8000 0000.

In the syntax, where *NSAT* appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This instruction is encoded as a **16-bit instruction** if the *NSAT* option is omitted. The 16-bit encoded instruction takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions. When the *NSAT* option is included, the instruction is encoded as a **32-bit instruction**. The 32-bit encoded instruction can sometimes save execution time (over a 16-bit encoded instruction), because it can be issued in parallel with certain other instructions.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Neg32 Example

```
r5 =-r0 ; /* default is no saturation */
r5 =-r0 (s) ; /* saturation */
r5 =-r0 (ns) ; /* no saturation */
```

Accumulator0 Negate (NegAcc0)

General Form

Dsp32Alu
a0 = -a0
a0 = -a1

Abstract

This instruction negates the input operands.

See Also (NegAcc1, NegAccDual)

NegAcc0 Description

The negate (two's-complement) instruction returns the same magnitude with the opposite arithmetic sign. The accumulator versions saturate the result at 40 bits. The instruction calculates by subtracting from zero.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	**AV0S**	**AV0**
...	...	AC1	**AC0**	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

NegAcc0 Example

```
a0 =-a0 ;
a0 =-a1 ;
```

Accumulator1 Negate (NegAcc1)

General Form

Dsp32Alu
a1 = -a0
a1 = -a1

Abstract

This instruction negates the input operands.

See Also (NegAcc0, NegAccDual)

NegAcc1 Description

The negate (two's-complement) instruction returns the same magnitude with the opposite arithmetic sign. The accumulator versions saturate the result at 40 bits. The instruction calculates by subtracting from zero.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	**AV1S**	**AV1**	AV0S	AV0
...	...	**AC1**	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

NegAcc1 Example

```
a1 =-a0 ;
a1 =-a1 ;
```

Dual Accumulator Negate (NegAccDual)

General Form

Dsp32Alu
a1 = -a1, a0 = -a0

Abstract

This instruction negates the input operands.

See Also (NegAcc0, NegAcc1)

NegAccDual Description

The dual negate (two's-complement) instruction returns the same magnitude with the opposite arithmetic sign for each accumulator. The accumulator versions saturate the result at 40 bits. The instruction calculates by subtracting from zero.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	**AC1**	**AC0**	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

NegAccDual Example

```
a1 =-a1, a0=-a0 ;
```

Fractional 32-bit to 16-Bit Conversion (Pass32Rnd16)

General Form

Dsp32Alu
DDST0_HL = DREG (rnd)

Abstract

This instruction converts a 32-bit, normalized-fraction number into a 16-bit normalized-fraction number by adding a round bit at bit 15, then saturating and extracting bits 31-16, then discarding bits 15-0. The instruction supports only biased rounding, which adds a half LSB (bit 15) before truncating bits 15-0. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

Pass32Rnd16 Description

The round to half-word instruction rounds a 32-bit, normalized-fraction number into a 16-bit, normalized-fraction number by extracting and saturating bits 31–16, then discarding bits 15–0. The instruction supports only biased rounding, which adds a half LSB (in this case, bit 15) before truncating bits 15–0. The ALU performs the rounding. The *RND_MOD* bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

Fractional data types such as the operands used in this instruction are always signed.

For more information, see the Saturation section and the Rounding and Truncation section in the Introduction chapter.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Pass32Rnd16 Example

```
   /* If r6 = 0xFFFC FFFF, then rounding to 16-bits with . . . */
r1.l = r6 (rnd) ;
   /* . . . produces r1.l = 0xFFFD */
   /* If r7 = 0x0001 8000, then rounding . . . */
r1.h = r7 (rnd) ;
   /* . . . produces r1.h = 0x0002 */
```

Accumulator0 32-Bit Saturate (ALU_SatAcc0)

General Form

Dsp32Alu
a0 = a0 (s)

Abstract

This instruciton saturates the accumulator at 32-bits (a0.w). The resulting saturated value is sign extended into the accumulator extension bits (a0.x).

See Also (ALU_SatAcc1, ALU_SatAccDual)

ALU_SatAcc0 Description

The saturate instruction saturates the 40-bit accumulators at 32 bits. The resulting saturated value is sign extended into the accumulator extension bits.

For more information, see the Saturation section in the Introduction chapter.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ALU_SatAcc0 Example

```
a0 = a0 (s) ;
```

Accumulator1 32-Bit Saturate (ALU_SatAcc1)

General Form

Dsp32Alu
a1 = a1 (s)

Abstract

This instruction saturates the accumulator at 32-bits (a1.w). The resulting saturated value is sign extended into the accumulator extension bits (a1.x).

See Also (ALU_SatAcc0, ALU_SatAccDual)

ALU_SatAcc1 Description

The saturate instruction saturates the 40-bit accumulators at 32 bits. The resulting saturated value is sign extended into the accumulator extension bits.

For more information, see the Saturation section in the Introduction chapter.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	**AV1S**	**AV1**	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ALU_SatAcc1 Example

```
a1 = a1 (s) ;
```

Dual Accumulator 32-Bit Saturate (ALU_SatAccDual)

General Form

Dsp32Alu
a1 = a1 (s), a0 = a0 (s)

Abstract

This instruction saturates both accumulators at 32-bits. The resulting saturated value is sign extended into the accumulator extension bits.

See Also (ALU_SatAcc0, ALU_SatAcc1)

ALU_SatAccDual Description

The dual saturate instruction saturates both 40-bit accumulators at 32 bits. The resulting saturated values are sign extended into the accumulator extension bits.

For more information, see the Saturation section in the Introduction chapter.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ALU_SatAccDual Example

```
a1 = a1 (s), a0 = a0 (s) ;
```

Logic Operations

These operations provide one's complement and other logic operations on register operands:

- 32-Bit One's Complement (Not32)
- 32-Bit Logic Operations (Logic32)

32-Bit Logic Operations (Logic32)

General Form

Comp3op
DREG = DREG & DREG
DREG = DREG | DREG
DREG = DREG ^ DREG

Abstract

This instruction performs logic operations on two 32-bit values. It does either an AND, OR, or XOR.

See Also (Not32)

Logic32 Description

The AND instruction performs a 32-bit, bit-wise logical AND operation on the two source registers and stores the results into the destination register. The instruction does not implicitly modify the source registers. The destination register and one source register can be the same D-register; this operation explicitly modifies the source register.

The OR instruction performs a 32-bit, bit-wise logical OR operation on the two source registers and stores the results into the destination register. The instruction does not implicitly modify the source registers. The destination register and one source register can be the same D-register; this operation explicitly modifies the source register.

The Exclusive-OR (XOR) instruction performs a 32-bit, bit-wise logical exclusive OR operation on the two source registers and loads the results into the destination register.

The XOR instruction does not implicitly modify source registers. The destination register and one source register can be the same D-register; this operation explicitly modifies the source register.

This **16-bit instruction** may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Logic32 Example

```
r4 = r4 & r3 ; /* AND */
r4 = r5 | r3 ; /* OR */
r4 = r2 ^ r4 ; /* XOR */
```

32-Bit One's Complement (Not32)

General Form

`ALU2op`
`DREG =~ DREG`

Abstract

This instruction (NOT one's complement) toggles every bit in the 32-bit register.

See Also (`Logic32`)

Not32 Description

The NOT one's-complement instruction toggles every bit in the 32-bit register. The instruction does not implicitly modify the source register. The destination register and source register can be the same D-register. Using the same D-register as the destination register and source register would explicitly modify the source register.

This **16-bit instruction** may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Not32 Example

```
r3 = ~ r4 ; /* NOT */
```

Move Operations

These operations provide register move operations on register, half register, and accumulator register operands:

- Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE)
- Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL)
- Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH)
- Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO)
- Move Register to Accumulator0 (MvAxToAx)
- Move Accumulator to Register (MvAxToDreg)
- Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL)
- Pass 8-Bit to 32-Bit Register Expansion (MvDregBToDreg)
- Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL)
- Move Register Half (LSBs) to 8-Bit Accumulator Section (MvDregLToAxX)
- Pass 16-Bit to 32-Bit Register Expansion (MvDregLToDreg)
- Move Register to Accumulator1 (MvDregToAx)
- Move Register to Accumulator0 & Accumulator1 (MvDregToAxDual)
- Move Register to Register (MvRegToReg)
- Conditional Move Register to Register (MvRegToRegCond)
- Dual Move Accumulators to Half Registers (ParaMvA1ToDregHwithMvA0ToDregL)
- Dual Move Accumulators to Register (ParaMvA1ToDregOwithMvA0ToDregE)

Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE)

General Form

Dsp32Mac
DREG_E = a0 MMODE

Abstract

This instruction moves an 32-bit section of an accumulator to an even register.

See Also (MvAxXToDregL, MvA0ToDregL, MvA1ToDregH, MvA0ToDregL, MvA1ToDregH, MvA1ToDreg0, MvAxToDreg, MvAxToAx)

MvA0ToDregE Description

The move accumulator register to even data register instruction copies the contents of the source accumulator register into the destination even data register. The operation does not affect the source register contents.

In the syntax, where *MMODE* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (is), (iss2), (iu), or (s2rnd). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvA0ToDregE Example

```
r2 = a0 ; /* 32-bit move with saturation */
r0 = a0 (iss2) ; /* 32-bit move with scaling, truncation and saturation */
```

Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL)

General Form

Dsp32Mac
DREG_L = a0 MMOD1

Abstract

This instruction moves an 16-bit section of an accumulator to a low half register (16-bit section of a data register).

See Also (MvAxXToDregL, MvA0ToDregL, MvA1ToDregH, MvA0ToDregE, MvA1ToDregH, MvA1ToDreg0, MvAxToDreg, MvAxToAx)

MvA0ToDregL Description

The move accumulator register to low half data register instruction copies the contents of the source accumulator register into the destination low half data register. The operation does not affect the source register contents.

In the syntax, where *MMOD1* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (ih), (is), (iss2), (iu), (s2rnd), (t), or (tfu). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvA0ToDregL Example

```
r3.l = a0 ;
r7.l = a0 (fu) ; /* fractional unsigned format */
r2.l = a0 (s2rnd) ; /* signed fraction, scaled */
```

Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH)

General Form

Dsp32Mac
DREG_H = a1 MMLMMOD1

Abstract

This instruction moves an 16-bit section of an accumulator to a high half register (16-bit section of a data register).

See Also (MvAxXToDregL, MvA0ToDregL, MvA1ToDregH, MvA0ToDregE, MvA0ToDregL, MvA1ToDreg0, MvAxToDreg, MvAxToAx)

MvA1ToDregH Description

The move accumulator register to low half data register instruction copies the contents of the source accumulator register into the destination low half data register. The operation does not affect the source register contents.

In the syntax, where *MMLMMOD1* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (ih), (is), (iss2), (iu), (m), (m,fu), (m,ih), (m,is), (m,iss2), (m,iu), (m,s2rnd), (m,t), (m,tfu), (s2rnd), (t), or (tfu). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvA1ToDregH Example

```
r3.h = a1 ;
r7.h = a1 (fu) ; /* fractional unsigned format */
r2.h = a1 (s2rnd) ; /* signed fraction, scaled */
```

Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO)

General Form

Dsp32Mac
DREG_O = a1 MMLMMODE

Abstract

This instruction moves an 32-bit section of an accumulator to an odd register.

See Also (MvAxXToDregL, MvA0ToDregL, MvA1ToDregH, MvA0ToDregE, MvA0ToDregL, MvA1ToDregH, MvAxToDreg, MvAxToAx)

MvA1ToDregO Description

The move accumulator register to low half data register instruction copies the contents of the source accumulator register into the destination low half data register. The operation does not affect the source register contents.

In the syntax, where *MMLMMODE* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (is), (iss2), (iu), (m), (m,fu), (m,is), (m,iss2), (m,iu), (m,s2rnd), (s2rnd) . See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvA1ToDregO Example

```
r3 = a1 ;
r7 = a1 (fu);
r1 = a1 (s2rnd);
```

Move Register to Accumulator0 (MvAxToAx)

General Form

Dsp32Alu
a0 = a1
a1 = a0

Abstract

This instruction moves the contents of one accumulator register to the other accumulator register.

See Also (MvAxXToDregL, MvA0ToDregL, MvA1ToDregH, MvA0ToDregE, MvA0ToDregL, MvA1ToDregH, MvA1ToDreg0, MvAxToDreg)

MvAxToAx Description

The move accumulator register to accumulator register instruction copies the contents of the source accumulator register into the destination accumulator register. The operation does not affect the source register contents.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

MvAxToAx Example

```
a0 = a1 ;
a1 = a0 ;
```

Move Accumulator to Register (MvAxToDreg)

General Form

Dsp32Mult
DREG_O = MUL1 MML , DREG_E = MUL0 MMODE
DREG = a1:0 M32MMOD2
DREG_PAIR = a1:0 M32MMOD

Abstract

This instruction moves the value in the accumulator register to the selected data register.

See Also (MvAxXToDregL, MvA0ToDregL, MvA1ToDregH, MvA0ToDregE, MvA0ToDregL, MvA1ToDregH, MvA1ToDreg0, MvAxToAx)

MvAxToDreg Description

The move accumulator register to low half data register instruction copies the contents of the source accumulator register into the destination low half data register. The operation does not affect the source register contents.

In the syntax, where *M32MMOD* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (is), (is,ns), (iu), (iu,ns), (m), (m,is), (m,is,ns), (m,t), (t), or (tfu). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvAxToDreg Example

```
r3 = a0 (iu,ns); /* integer unsigned no saturate */
r4 = a1 (fu); /* fractional unsigned */
r2 = a0 (m); /* mixed mode */
r5 = a1 (tfu); /* fractional unsigned truncated */
```

Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL)

General Form

Dsp32Alu
DREG_L = a0.x
DREG_L = a1.x

Abstract

This instruction moves an 8-bit section of an accumulator to a low half register (16-bit section of a data register).

See Also (MvA0ToDregL, MvA1ToDregH, MvA0ToDregE, MvA0ToDregL, MvA1ToDregH, MvA1ToDreg0, MvAx-ToDreg, MvAxToAx)

MvAxXToDregL Description

The move accumulator register extension copies 8 bits from an accumulator extension source register into a low half data register. The instruction does not affect the unspecified half of the destination register. It supports only data registers and the accumulator.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

MvAxXToDregL Example

```
r7.l = a0.x ;
r0.l = a1.x ;
```

Pass 8-Bit to 32-Bit Register Expansion (MvDregBToDreg)

General Form

ALU2op
DREG = DREG_B (x)
DREG = DREG_B (z)

Abstract

This instruction copies the least significant 8-bits from the source register into the least significant 8-bits of the destination and either sign or zero extends it the upper bits.

See Also (MvDregLToAxX, MvDregHLToAxHL, MvDregLToDreg, MvDregToAx, MvDregToAxDual)

MvDregBToDreg Description

The move data register byte to data register instruction converts a signed byte to a signed word (32 bits). It copies the least significant 8 bits from a source register into the least significant 8 bits of a 32-bit register. The instruction sign-extends or zero-extends the upper bits of the destination register. This instruction supports only data registers.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), and this instruction may be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvDregBToDreg Example

```
r7 = r2.b (x) ; /* sign extended */
r7 = r2.b (z) ; /* zero extended */
```

Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL)

General Form

Dsp32Alu
A0_HL = DSRC0_HL
A1_HL = DSRC0_HL

Abstract

This instruction moves a 16-bit section of a data register to a single 16-bit section of an accumulator.

See Also (MvDregLToAxX, MvDregLToDreg, MvDregBToDreg, MvDregToAx, MvDregToAxDual)

MvDregHLToAxHL Description

The move high/low half register to high/low half accumulator instruction copies 16 bits from a source register into half of an accumulator register. The instruction does not affect the unspecified half of the destination register. It supports only data registers and the accumulator.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

MvDregHLToAxHL Example

```
a0.l = r0.l /* least significant 16 bits of Dreg into least significant 16 bits of A0.
W */
a0.h = r1.l
a1.l = r2.l /* least significant 16 bits of Dreg into least significant 16 bits of A1.
W */
a1.h = r3.l
a0.l = r4.h
a0.h = r5.h /* most significant 16 bits of Dreg into most significant 16 bits of A0.W */
a1.l = r6.h
a1.h = r7.h /* most significant 16 bits of Dreg into most significant 16 bits of A1.W */
```

Move Register Half (LSBs) to 8-Bit Accumulator Section (MvDregLToAxX)

General Form

Dsp32Alu
a0.x = DREG_L
a1.x = DREG_L

Abstract

This instruction moves the 8 LSBs from a low half register (16-bit section of a data register) to an 8-bit section of an accumulator.

See Also (MvDregHLToAxHL, MvDregLToDreg, MvDregBToDreg, MvDregToAx, MvDregToAxDual)

MvDregLToAxX Description

The move low half data register to accumulator register extension instruction copies 8 bits from a low half data register source into an accumulator extension register. The instruction does not affect the unspecified portion of the destination register. It supports only data registers and the accumulator.

The accumulator extension registers A0.X and A1.X are defined only for the 8 low-order bits 7 through 0 of A0.X and A1.X. This instruction truncates the upper byte of *DDSTO_L* before moving the value into the accumulator extension register (A0.X or A1.X).

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

MvDregLToAxX Example

```
a0.x = r1.l ;
a1.x = r4.l ;
```

Pass 16-Bit to 32-Bit Register Expansion (MvDregLToDreg)

General Form

ALU2op
DREG = DREG_L (x)
DREG = DREG_L (z)

Abstract

This instruction zero extends or sign extends a 16-Bit register half and deposits it into a 32-bit destination register. The X option signifies sign extension signifies sign extension while the Z signifies zero extension.

See Also (MvDregLToAxX, MvDregHLToAxHL, MvDregBToDreg, MvDregToAx, MvDregToAxDual)

MvDregLToDreg Description

The move low half register to data register instruction converts an unsigned half word (16 bits) to an unsigned word (32 bits). The instruction copies the least significant 16 bits from a source register into the lower half of a 32-bit register and sign- or zero-extends the upper half of the destination register. The operation supports only data registers. Zero extension is appropriate for unsigned values. If used with signed values, a small negative 16-bit value will become a large positive value.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), and this instruction may be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvDregLToDreg Example

```
/* If r0.l = 0xFFFF, before move with zero extend ... */
r4 = r0.l (z) ; /* zero-extends; equivalent operation to r4.l = r0.l and r4.h = 0 */
/* . . . then r4 = 0x0000FFFF, after move with zero extend */
r4 = r0.l (x) ; /* sign-extends */
```

Move Register to Accumulator1 (MvDregToAx)

General Form

Dsp32Alu
a0 = DREG XMODE
a1 = DREG XMODE

Abstract

This instruction moves the contents of a data register to an accumulator register.

See Also (MvDregLToAxX, MvDregHLToAxHL, MvDregLToDreg, MvDregBToDreg, MvDregToAxDual)

MvDregToAx Description

The move data register to accumulator instruction copies 32 bits from a source register into Ax.W section of an accumulator register with zero- or sign-extension. The instruction does not affect the unspecified portion of the destination register. It supports only data registers and the accumulator.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

MvDregToAx Example

```
a0 = r7 (z) ; /* move R7 to 32-bit A0.W, zero extended */
a1 = r3 (x) ; /* move R3 to 32-bit A1.W, sign-exteneded */
```

Move Register to Accumulator0 & Accumulator1 (MvDregToAxDual)

General Form

Dsp32Alu
a1 = DREG SMODE , a0 = DREG XMODE

Abstract

This instruction moves the contents of two data registers to the accumulator registers (A0, A1).

See Also (MvDregLToAxX, MvDregHLToAxHL, MvDregLToDreg, MvDregBToDreg, MvDregToAx)

MvDregToAxDual Description

The dual move data register to accumulator instruction copies 32 bits from a source register into Ax.W section of an accumulator register with zero- or sign-extension, and perform a second move in parallel as indicated. The instruction does not affect the unspecified portion of the destination register. It supports only data registers and the accumulator.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

MvDregToAxDual Example

```
a1 = r3 (z), a0 = r7 (z) ; /* move R3 to 32-bit A1.W (zero extended), move R7 to 32-
bit A0.W (zero extended) */
a1 = r3 (z), a0 = r7 (x) ; /* move R3 to 32-bit A1.W (zero extended), move R7 to 32-
bit A0.W (sign-exteneded) */
a1 = r3 (x), a0 = r7 (z) ; /* move R3 to 32-bit A1.W (sign-exteneded), move R7 to 32-
bit A0.W (zero extended) */
a1 = r3 (x), a0 = r7 (x) ; /* move R3 to 32-bit A1.W (sign-exteneded), move R7 to 32-
bit A0.W (sign-exteneded) */
```

Move Register to Register (MvRegToReg)

General Form

RegMv
GDST = GSRC

Abstract

This instruction moves data from any register in the data arithmetic unit, address arithmetic unit, or control unit to any other register in those units.

See Also (MvRegToRegCond)

MvRegToReg Description

The move any register to any register instruction copies from a source register into a destination register with zero- or sign-extension. The instruction does not affect the unspecified portion of the destination register. It supports all processor core registers. All moves from smaller to larger registers are sign extended.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), and this instruction may be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either **User or Supervisor mode**, except for cases where register access restrictions only permit **Supervisor mode** .

MvRegToReg Example

```
r3 = r0 ;
r7 = p2 ;
r2 = a0 ;
a0.w = r7 ; /* move R7 to 32-bit A0.W */
r3 = a1.x ; /* move 8-bit A1.X to R3 with sign extension*/
retn = p0 ; /* must be in Supervisor mode */
r7 = a0 ; /* move A0 to odd data register */
r2 = a1 ; /* move A1 to even data register */
```

Conditional Move Register to Register (MvReg-ToRegCond)

General Form

CCMV
if cc GDST = GSRC
if !cc GDST = GSRC

Abstract

This instruction conditionally moves registers.

See Also (MvRegToReg)

MvRegToRegCond Description

The Move Conditional instruction moves source register contents into a destination register, depending on the value of CC.

- `IF CC` *`DPreg = DPreg`*, the move occurs only if CC = 1.
- `IF ! CC` *`DPreg = DPreg`*, the move occurs only if CC = 0.

The source and destination registers are any data register or pointer register.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

MvRegToRegCond Example

```
if cc r3 = r0 ; /* move if CC=1 */
if cc r2 = p4 ;
if cc p0 = r7 ;
if cc p2 = p5 ;
if ! cc r3 = r0 ; /* move if CC=0 */
if ! cc r2 = p4 ;
if ! cc p0 = r7 ;
if ! cc p2 = p5 ;
```

Dual Move Accumulators to Half Registers (ParaMvA1ToDregHwithMvA0ToDregL)

General Form

Dsp32Mac
DREG_H = a1 MML , DREG_L = a0 MMOD1

Abstract

This dual move instruction moves the contents of the accumulator registers to half data registers.

See Also (ParaMvA1ToDregOwithMvA0ToDregE)

ParaMvA1ToDregHwithMvA0ToDregL Description

The dual move accumulator 1 to high half register with move accumulator 0 to low half register instruction provide a the combination of operations in the MvA0ToDregL and MvA1ToDregH.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMvA1ToDregHwithMvA0ToDregL Example

```
r3.h = a1 , r3.l = a0 ;
r2.h = a1 (s2rnd) , r7.l = a0 (fu) ; /* signed fraction, scaled, fractional unsigned
format */
r7.h = a1 (fu) , r2.l = a0 (s2rnd) ; /* fractional unsigned format, signed fraction,
scaled */
```

Dual Move Accumulators to Register (ParaMvA1ToDregOwithMvA0ToDregE)

General Form

Dsp32Mac
DREG_O = a1 MML , DREG_E = a0 MMODE

Abstract

This dual move instruction moves the contents of the accumulator registers to data registers.

See Also (ParaMvA1ToDregHwithMvA0ToDregL)

ParaMvA1ToDregOwithMvA0ToDregE Description

The dual move accumulator 1 to high half register with move accumulator 0 to low half register instruction provide a the combination of operations in the MvA1ToDreg0 with MvA0ToDregE.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMvA1ToDregOwithMvA0ToDregE Example

```
r3 = a1 , r0 = a0 (iss2) ;
r7 = a1 (fu) , r2 = a0 ;
r1 = a1 (s2rnd) , r2 = a0 ;
```

Multiplication Operations

These operations provide multiply and multiply-accumulate operations on register and immediate value operands:

- 16 x 16-Bit MAC (Mac16)
- 16 x 16-Bit MAC with Move to Register (Mac16WithMv)
- 32 x 32-Bit MAC (Mac32)
- 32 x 32-Bit MAC with Move to Register (Mac32WithMv)
- Complex Multiply to Accumulator (Mac32Cmplx)
- Complex Multiply to Register (Mac32CmplxWithMv)
- Complex Multiply to Register with Narrowing (Mac32CmplxWithMvN)
- 16 x 16-Bit Multiply (Mult16)
- 32 x 32-bit Multiply (Mult32)
- 32 x 32-Bit Multiply, Integer (MultInt)
- Dual 16 x 16-Bit MAC (ParaMac16AndMac16)
- Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMac16WithMv)
- Dual 16 x 16-Bit MAC with Move to Register (ParaMac16WithMvAndMac16)
- Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMac16WithMv)
- Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMv)
- Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMv)
- Dual 16 x 16-Bit Multiply (ParaMult16AndMult16)
- Dual Move to Register and 16 x 16-Bit MAC (ParaMvAndMac16)
- Dual Move to Register and 16 x 16-Bit MAC with Move to Register (ParaMvAndMac16WithMv)

16 x 16-Bit MAC (Mac16)

General Form

Dsp32Mac
MAC0 MMOD0
MAC0 MMOD0
MAC1 MMLMMOD0
MAC1 MMLMMOD0

Abstract

This multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required.

See Also (Mac16WithMv)

Mac16 Description

The Multiply and Multiply-Accumulate to Accumulator instruction multiplies two 16-bit half-word operands. It stores, adds or subtracts the product into a designated Accumulator with saturation.

The Multiply-and-Accumulate Unit 0 (MAC0) portion of the architecture performs operations that involve Accumulator A0. MAC1 performs A1 operations.

By default, the instruction treats both operands of both MACs as signed fractions with left-shift correction as required.

In the syntax, where *MMOD0* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, or `(w32)`.

In the syntax, where *MMLMMOD0* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(m)`, `(W32)`, `(m,fu)`, `(m,is)`, or `(m,w32)`.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac16 Example

```
a0 = r3.h * r2.h ; /* MAC0, only. Both operands are signed fractions. Load the product
into A0. */
a0 += r6.h * r4.l (fu) ; /* MAC0, only. Both operands are unsigned fractions.
Accumulate into A0 */
a0 -= r3.h * r2.h ; /* MAC0, only. Both operands are signed fractions.  Accumulate
into A0. */
a1 = r6.h * r4.l (fu) ; /* MAC1, only. Both operands are unsigned fractions. Load the
product into A1 */
a1 += r3.h * r2.h ; /* MAC1, only. Both operands are signed fractions. Accumulate into
A1. */
a1 -= r6.h * r4.l (fu) ; /* MAC1, only. Both operands are unsigned fractions.
Accumulate into A1 */
```

16 x 16-Bit MAC with Move to Register (Mac16WithMv)

General Form

Dsp32Mac
DREG_L =(MAC0) MMOD1
DREG_L =(MAC0) MMOD1
DREG_H =(MAC1) MMLMMOD1
DREG_H =(MAC1) MMLMMOD1

Abstract

This multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required.

See Also (Mac16)

Mac16WithMv Description

The multiply and multiply-accumulate to half register (with move) instruction multiplies two 16-bit half-word operands. The instruction stores, adds or subtracts the product into a designated accumulator. Then, it copies 16 bits (saturated at 16 bits) of the accumulator into a high or low half data register.

In the syntax, where *MMOD1* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (ih), (is), (iss2), (iu), (s2rnd), (t), (tfu) .

In the syntax, where *MMLMMOD1* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (ih), (is), (iss2), (iu), (m), (m,fu), (m,ih), (m,is), (m,iss2), (m,iu), (m,s2rnd), (m,t), (m,tfu), (s2rnd), (t), (tfu) .

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

The fraction versions of this instruction (the default and (fu) options) transfer the Accumulator result to the destination register according to the **Result to Destination Register ((IS) and (IU) Options)** diagram.

The integer versions of this instruction (the (is) and (iu) options) transfer the Accumulator result to the destination register according to the **Result to Destination Register ((IS) and (IU) Options)** diagram.

The multiply-and-accumulate unit 0 (MAC0) portion of the architecture performs operations that involve accumulator A0 and loads the results into the lower half of the destination data register. MAC1 performs A1 operations and loads the results into the upper half of the destination data register.

All versions of this instruction that support rounding are affected by the RND_MOD bit in the ASTAT register when they copy the results into the destination register. RND_MOD determines whether biased or unbiased rounding is used.

Figure 8-4: Result to Destination Register (Default and (FU) Options)

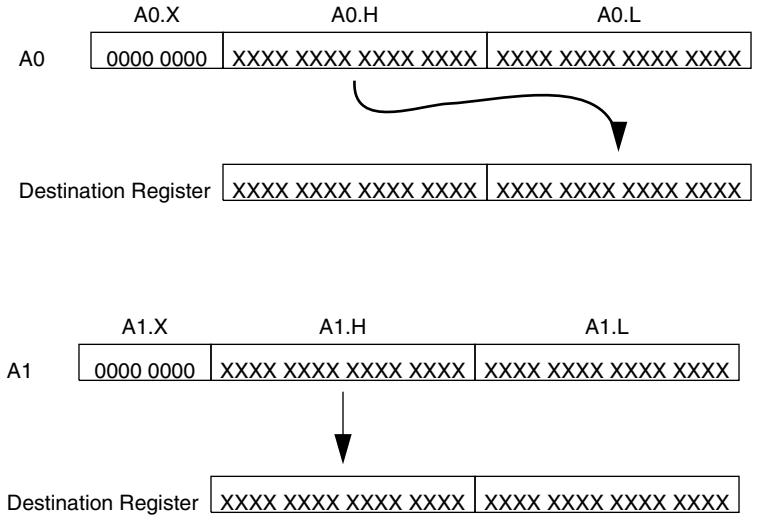
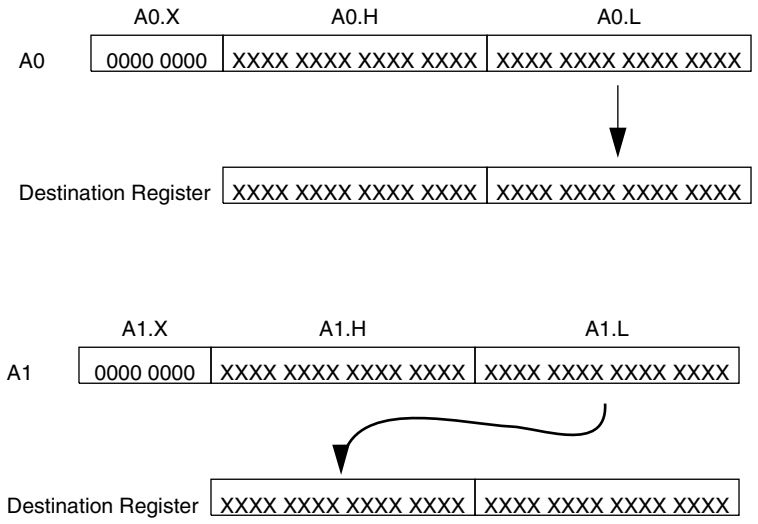


Figure 8-5: Result to Destination Register ((IS) and (IU) Options)



This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

This instruction has **special applications**. DSP filter applications often use the multiply and multiply-accumulate to half-register instruction to calculate the dot product between two signal vectors.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac16WithMvExample

```
r3.l = ( a0 = r3.h * r2.h ) ;
/* MAC0, only. Both operands are signed fractions. Load the product into A0, then copy
to r3.l. */
r3.h = ( a1 += r6.h * r4.l ) (fu) ;
/* MAC1, only. Both operands are unsigned fractions. Add the product into A1, then
copy to r3.h */
```

32 x 32-Bit MAC (Mac32)

General Form

Dsp32Mult
a1:0 = DREG * DREG M32MMOD
a1:0 += DREG * DREG M32MMOD
a1:0 -= DREG * DREG M32MMOD

Abstract

This instruction executes a multiply accumulate operation on 32-bit registers.

See Also (Mac32WithMv)

Mac32 Description

The multiply-accumulate to accumulator instruction multiplies two 32-bit half-word operands. It stores, adds or subtracts the product into a designated accumulator with saturation.

The multiply-and-accumulate unit 0 (MAC0) portion of the architecture performs operations that involve Accumulator A0. MAC1 performs A1 operations.

By default, the instruction treats both operands of both MACs as signed fractions with left-shift correction as required.

In the syntax, where *M32MMOD0* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (is), (is,ns), (iu), (iu,ns), (m), (m,is), or (m,is,ns).

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac32 Example

```
a0 = r0 * r7 ; /* (default) fractional signed, place product in a0 */
a0 += r1 * r6 (fu) ; /* fractional unsigned, accumulate in a0 */
a0 -= r2 * r5 (is) ; /* integer signed, accumulate in a0 */
a1 = r3 * r4 (is,ns) ; /* integer signed (no saturation), place product in a1 */
a1 += r4 * r3 (iu) ; /* integer unsigned, accumulate in a1 */
a1 -= r5 * r2 (iu,ns) ; /* integer unsigned (no saturation), accumulate in a1 */
a0 = r6 * r1 (m) ; /* mixed mode and fractional signed, place product in a0 */
a0 += r7 * r0 (m,is) ; /* mixed mode and integer signed, accumulate in a0 */
a0 -= r0 * r7 (m,is,ns) ; /* mixed mode and integer signed (no saturation), accumulate
in a0 */
a1 = r1 * r6 ; /* (default) fractional signed, place product in a1 */
a1 += r2 * r5 (fu) ; /* fractional unsigned, accumulate in a1 */
a1 -= r3 * r4 (is) ; /* integer signed, accumulate in a1 */
```

32 x 32-Bit MAC with Move to Register (Mac32WithMv)

General Form

Dsp32Mac
DREG_E =(MAC0) MMODE
DREG_E =(MAC0) MMODE
DREG_O =(MAC1) MMLMMODE
DREG_O =(MAC1) MMLMMODE
Dsp32Mult
DREG =(a1:0 = DREG * DREG) M32MMOD1
DREG =(a1:0 += DREG * DREG) M32MMOD1
DREG =(a1:0 -= DREG * DREG) M32MMOD1
DREG_PAIR =(a1:0 = DREG * DREG) M32MMOD
DREG_PAIR =(a1:0 += DREG * DREG) M32MMOD
DREG_PAIR =(a1:0 -= DREG * DREG) M32MMOD

Abstract

This multiply-accumulate instruction multiplies two 32-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required.

See Also (Mac32)

Mac32WithMv Description

The multiply-accumulate to accumulator instruction multiplies two 32-bit half-word operands. It stores, adds or subtracts the product into a designated accumulator with saturation. Then, the instruction moves the result to the selected register or register pair.

The multiply-and-accumulate unit 0 (MAC0) portion of the architecture performs operations that involve Accumulator A0. MAC1 performs A1 operations.

By default, the instruction treats both operands of both MACs as signed fractions with left-shift correction as required.

In the syntax, where *MMODE* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (is), (iss2), (iu), or (s2rnd).

In the syntax, where *MMLMMODE* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(iss2)`, `(iu)`, `(m)`, `(m,fu)`, `(m,is)`, `(m,iss2)`, `(m,iu)`, `(m,s2rnd)`, or `(s2rnd)`.

In the syntax, where *M32MMOD1* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(is,ns)`, `(iu)`, `(iu,ns)`, `(m,is)`, `(m,is,ns)`, `(m,t)`, `(t)`, or `(tfu)`.

In the syntax, where *M32MMOD* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(is,ns)`, `(iu)`, `(iu,ns)`, `(m)`, `(m,is)`, or `(m,is,ns)`.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac32WithMv Example

```
r0 = (a0 = r0 * r3) (fu) ; /* MMODE options, place product in a0 and move it to EVEN
data register */
r2 = (a0 += r1 * r2) (is) ; /* MMODE options, accumulate in a0 and move it to EVEN
data register */
r4 = (a0 -= r2 * r1) (iss2) ; /* MMODE options, accumulate in a0 and move it to EVEN
data register */
r1 = (a1 = r3 * r0) (m,fu) ; /* MMLMMODE options, place product in a1 and move it to
ODD data register */
r3 = (a1 += r4 * r7) (m,is) ; /* MMLMMODE options, accumulate in a1 and move it to ODD
data register */
r5 = (a1 -= r5 * r6) (m,s2rnd) ; /* MMLMMODE options, accumulate in a1 and move it to
ODD data register */
r1 = (a0 = r6 * r5) (t) ; /* M32MMOD1 options, place product in a0 and move it to any
data register */
r2 = (a0 += r7 * r4) (tfu) ; /* M32MMOD1 options, accumulate in a0  and move it to any
data register*/
r3 = (a0 -= r0 * r3) (m,is,ns) ; /* M32MMOD1 options, accumulate in a0  and move it
to any data register*/
```

```
r4 = (a1 = r1 * r2) (iu,ns) ; /* M32MMOD1 options, place product in a1  and move it
to any data register*/
r5 = (a1 += r2 * r1) (fu) ; /* M32MMOD1 options, accumulate in a1  and move it to any
data register*/
r6 = (a1 -= r3 * r0) (m,is) ; /* M32MMOD1 options, accumulate in a1  and move it to
any data register*/
r1:0 = (a0 = r4 * r7) (fu) ; /* M32MMOD options, place product in a0 and move it to
register pair */
r3:2 = (a0 += r5 * r6) ; /* M32MMOD options, accumulate in a0 and move result to
register pair */
r5:4 = (a0 -= r6 * r5) (m) ; /* M32MMOD options, accumulate in a0 and move result to
register pair */
r7:6 = (a1 = r7 * r4) (is,ns) ; /* M32MMOD options, place product in a1 and move it
to register pair */
r5:4 = (a1 += r0 * r3) (iu,ns) ; /* M32MMOD options, accumulate in a1 and move result
to register pair */
r3:2 = (a1 -= r1 * r2) (is) ; /* M32MMOD options, accumulate in a1 and move result to
register pair */
```

Complex Multiply to Accumulator (Mac32Cmplx)

General Form

Dsp32Mac
a1:0 = CMPLXOP CMODE
a1:0 += CMPLXOP CMODE
a1:0 -= CMPLXOP CMODE

Abstract

This instruction executes a complex multiply-accumulate operation, placing the results in an accumulator register.

See Also (Mac32CmplxWithMv, Mac32CmplxWithMvN)

Mac32Cmplx Description

The multiply-accumulate complex values instruction performs a number of parallell multiply-accumulate operations to produce complex results. To understand the operations, it is important to understand the placement of the imaginary part and real part of the data. Let operand A = (A_r + j *B_i), operand B = (B_r + j *B_i) and result C = (C_r + j *C_i), where A_i (the imaginary part) is stored in the most significant 16 bits of a 32 bit register, and A_r (the real part) is stored in the least significant 16 bits. Other notations (such as B_i, C_i, and others) are similarly defined regarding data placement of imaginary and real parts. Complex multiplication and complex multiplication of conjugates is defined as follows:

Table 8-6: Complex Multiplication and Complex Conjugates

Table 8-7:

Complex Multiplication	Imaginary Conjugate	Real Conjugate
C = cmul(A, B)	$C_i = A_r*B_i + A_i*B_r$	$C_r = A_r*B_r - A_i*B_i$
C = cmul(A, B*)	$C_i = A_i*B_r - A_r*B_i$	$C_r = A_r*B_r + A_i*B_i$
C = cmul(A*, B*)	$C_i = -(A_r*B_i + A_i*B_r)$	$C_r = A_r*B_r - A_i*B_i$

This complex multiply syntax for placing the product in the accumulator registers corresponds to the commented operations:

```
a1:0 = cmul(r1,r0); /* complex multiply of r1 and r0, place imaginary product in a1
and real product in a0 */
/* a1 = (r1.l * r0.h) + (r1.h * r0.l), a0 = (r1.l * r0.l) - (r1.h * r0.h) */

a1:0 = cmul(r1,r0*); /* complex multiply of r1 and r0, place imaginary product in a1
and real product in a0 */
```

```
/* a1 = (r1.h * r0.l) - (r1.l * r0.h), a0 = (r1.l * r0.l) + (r1.h * r0.h) */

a1:0 = cmul(r1*,r0*); /* complex multiply of r1 and r0, place imaginary product in a1
and real product in a0 */
/* a1 = - [ (r1.l * r0.h) + (r1.h * r0.l) ], a0 = (r1.l * r0.l) - (r1.h * r0.h) */
```

In the syntax, where *CMODE* appears, substitute a complex multiply mode for the accumulator copy format option: default (none) or `(is)`.

Default operation is **signed fraction multiplication**. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction for each of the four partial products. Add or subtract corresponding partial products for real and imaginary part of result. Saturate results between minimum -1 and maximum 1^{-2-31}. The resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF. This operation uses **signed fraction rounding**.

If the `(is)` option is used, the operation is **signed integer multiplication**. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Saturate integer results between minimum -2^{31} and maximum $2^{31}-1$.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac32Cmplx Example

```
a1:0 = (r1,r0) ; /* fractional signed complex multiply; place complex product in a1
(imaginary) and a0 (real) */
a1:0 = (r7,r3*) (is) ; /* integer signed complex multiply; place complex product in
a1 (imaginary) and a0 (real) */
a1:0 = (r2*,r4*) ; /* fractional signed complex multiply; place complex product in a1
(imaginary) and a0 (real) */
a1:0 += (r3*,r1*) (is) ;  /* integer signed complex mac; accumulate complex result in
a1 (imaginary) and a0 (real) */
a1:0 += (r5,r2*) ; /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real) */
```

```
a1:0 += (r6,r1) ; /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real) */
a1:0 -= (r2,1) (is) ;  /* integer signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real) */
a1:0 -= (r3*,r7*) /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real) */
a1:0 -= (r4,r5*) (is) ; /* integer signed complex mac; accumulate complex result in
a1 (imaginary) and a0 (real) */
```

Complex Multiply to Register (Mac32Cmplx-WithMv)

General Form

Dsp32Mac
DREG_PAIR = CMPLXOP CMODE
DREG_PAIR = (a1:0 = CMPLXOP) CMODE
DREG_PAIR = (a1:0 += CMPLXOP) CMODE
DREG_PAIR = (a1:0 -= CMPLXOP) CMODE

Abstract

This instruction executes a complex multiply-accumulate operation, placing the results in a register or register pair.

See Also (Mac32Cmplx, Mac32CmplxWithMvN)

Mac32CmplxWithMv Description

The multiply-accumulate complex values instruction performs a number of parallell multiply-accumulate operations to produce complex results with a move. The product of the multiplication is placed in a pair of data registers. Alternately, the instruction may accumulate the result in the accumulator registers, then move the result to a pair of data registers. To understand the operations, it is important to understand the placement of the imaginary part and real part of the data. Let operand A = (A_r + j *B_i), operand B = (B_r + j *B_i) and result C = (C_r + j *C_i), where A_i (the imaginary part) is stored in the most significant 16 bits of a 32 bit register, and A_r (the real part) is stored in the least significant 16 bits. Other notations (such as B_i, C_i, and others) are similarly defined regarding data placement of imaginary and real parts. Complex multiplication and complex multiplication of conjugates is defined as follows:

Table 8-8: Complex Multiplication and Complex Conjugates

Table 8-9:

Complex Multiplication	Imaginary Conjugate	Real Conjugate
C = cmul(A, B)	$C_i = A_r*B_i + A_i*B_r$	$C_r = A_r*B_r - A_i*B_i$
C = cmul(A, B*)	$C_i = A_i*B_r - A_r*B_i$	$C_r = A_r*B_r + A_i*B_i$
C = cmul(A*, B*)	$C_i = -(A_r*B_i + A_i*B_r)$	$C_r = A_r*B_r - A_i*B_i$

This complex multiply syntax for placing the product in the accumulator registers corresponds to the commented operations:

```
a1:0 = cmul(r1,r0); /* complex multiply of r1 and r0, place imaginary product in a1
and real product in a0 */
/* a1 = (r1.l * r0.h) + (r1.h * r0.l), a0 = (r1.l * r0.l) - (r1.h * r0.h) */

a1:0 = cmul(r1,r0*); /* complex multiply of r1 and r0, place imaginary product in a1
and real product in a0 */
/* a1 = (r1.h * r0.l) - (r1.l * r0.h), a0 = (r1.l * r0.l) + (r1.h * r0.h) */

a1:0 = cmul(r1*,r0*); /* complex multiply of r1 and r0, place imaginary product in a1
and real product in a0 */
/* a1 = - [ (r1.l * r0.h) + (r1.h * r0.l) ], a0 = (r1.l * r0.l) - (r1.h * r0.h) */
```

In the syntax, where *CMODE* appears, substitute a complex multiply mode for the accumulator copy format option: default (none) or `(is)`.

Default operation is **signed fraction multiplication**. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction for each of the four partial products. Add or subtract corresponding partial products for real and imaginary part of result. Saturate results between minimum -1 and maximum 1^{-2-31}. The resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF.

If the `(is)` option is used, the operation is **signed integer multiplication**. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Saturate integer results between minimum -2^{31} and maximum $2^{31}-1$.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac32CmplxWithMv Example

```
r7:6 = (r1,r0) ; /* fractional signed complex multiply; place complex product in r7
(imaginary) and r6 (real) */
r5:4 = (r7,r3*) (is) ; /* integer signed complex multiply; place complex product in
r5 (imaginary) and r4 (real) */
r1:0 = (r2*,r4*) ; /* fractional signed complex multiply; place complex product in r1
(imaginary) and r0 (real) */
r7:6 = a1:0 = (r1,r0) ; /* fractional signed complex multiply; place complex product
in a1 (imaginary) and a0 (real); move to r7:6 */
r5:4 = a1:0 = (r7,r3*) (is) ; /* integer signed complex multiply; place complex product
in a1 (imaginary) and a0 (real); move to r5:4 */
r1:0 = a1:0 = (r2*,r4*) ; /* fractional signed complex multiply; place complex product
in a1 (imaginary) and a0 (real); move to r1:0 */
r5:4 = a1:0 += (r3*,r1*) (is) ;  /* integer signed complex mac; accumulate complex
result in a1 (imaginary) and a0 (real); move to r5:4 */
r1:0 = a1:0 += (r5,r2*) ; /* fractional signed complex mac; accumulate complex result
in a1 (imaginary) and a0 (real); move to r1:0 */
r3:2 = a1:0 += (r6,r1) ; /* fractional signed complex mac; accumulate complex result
in a1 (imaginary) and a0 (real); move to r3:2 */
r7:6 = a1:0 -= (r2,1) (is) ;  /* integer signed complex mac; accumulate complex result
in a1 (imaginary) and a0 (real); move to r7:6 */
r1:0 = a1:0 -= (r3*,r7*) /* fractional signed complex mac; accumulate complex result
in a1 (imaginary) and a0 (real); move to r1:0 */
r3:2 = a1:0 -= (r4,r5*) (is) ; /* integer signed complex mac; accumulate complex result
in a1 (imaginary) and a0 (real); move to r3:2 */
```

Complex Multiply to Register with Narrowing (Mac32CmplxWithMvN)

General Form

Dsp32Mac
DREG = CMPLXOP NARROWING_CMODE
DREG = (a1:0 = CMPLXOP) NARROWING_CMODE
DREG = (a1:0 += CMPLXOP) NARROWING_CMODE
DREG = (a1:0 -= CMPLXOP) NARROWING_CMODE

Abstract

This instruction executes a complex multiply-accumulate operation, placing the results in a register or register pair with narrowing.

See Also (Mac32Cmplx, Mac32CmplxWithMv)

Mac32CmplxWithMvN Description

The multiply-accumulate complex values instruction performs a number of parallell multiply-accumulate operations to produce complex results with a narrowing move. The product of the multiplication is placed in a data register. Alternately, the instruction may accumulate the result in the accumulator registers, then move the result to a data register. To understand the operations, it is important to understand the placement of the imaginary part and real part of the data. Let operand A = (A_r + j *B_i), operand B = (B_r + j *B_i) and result C = (C_r + j *C_i), where A_i (the imaginary part) is stored in the most significant 16 bits of a 32 bit register, and A_r (the real part) is stored in the least significant 16 bits. Other notations (such as B_i, C_i, and others) are similarly defined regarding data placement of imaginary and real parts. Complex multiplication and complex multiplication of conjugates is defined as follows:

Table 8-10: Complex Multiplication and Complex Conjugates

Table 8-11:

Complex Multiplication	Imaginary Conjugate	Real Conjugate
C = cmul(A, B)	$C_i = A_r*B_i + A_i*B_r$	$C_r = A_r*B_r - A_i*B_i$
C = cmul(A, B*)	$C_i = A_i*B_r - A_r*B_i$	$C_r = A_r*B_r + A_i*B_i$
C = cmul(A*, B*)	$C_i = -(A_r*B_i + A_i*B_r)$	$C_r = A_r*B_r - A_i*B_i$

This complex multiply syntax for placing the product in the accumulator registers corresponds to the commented operations:

```
a1:0 = cmul(r1,r0); /* complex multiply of r1 and r0, place imaginary product in a1
and real product in a0 */
/* a1 = (r1.l * r0.h) + (r1.h * r0.l), a0 = (r1.l * r0.l) - (r1.h * r0.h) */

a1:0 = cmul(r1,r0*); /* complex multiply of r1 and r0, place imaginary product in a1
and real product in a0 */
/* a1 = (r1.h * r0.l) - (r1.l * r0.h), a0 = (r1.l * r0.l) + (r1.h * r0.h) */

a1:0 = cmul(r1*,r0*); /* complex multiply of r1 and r0, place imaginary product in a1
and real product in a0 */
/* a1 = - [ (r1.l * r0.h) + (r1.h * r0.l) ], a0 = (r1.l * r0.l) - (r1.h * r0.h) */
```

In the syntax, where *NARROWING_CMODE* appears, substitute a complex multiply mode for the accumulator copy format option: default (none), `(is)`, or `(t)`.

Default operation is **signed fraction multiplication**. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction for each of the four partial products. Add or subtract corresponding partial products for real and imaginary part of result. Saturate results between minimum -1 and maximum 1^{-2-31}. The resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF. This operation uses **signed fraction rounding**. Round 1.31 format value at bit 16, (RND_MOD bit in the ASTAT register controls the rounding) extract the high 16 bits to produce a 1.15 result. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).

If the `(is)` option is used, the operation is **signed integer multiplication**. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Saturate integer results between minimum -2^{31} and maximum 2^{31}-1. This operation uses **signed integer saturation**. Saturate 32-bit integer values at bit 15 and extract the low 16 bits to produce a result between minimum -2^{15} and maximum 2^{15}-1.

If the `(t)` option is used, the operation is **signed fraction multiplication with truncation**. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction for each of the four partial products. Add or subtract corresponding partial products for real and imaginary part of result. Saturate results between minimum -1 and maximum 1^{-2-31}. The resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF. This operation uses **signed fraction truncation**. Truncate 1.31 format values for real and imaginary parts of the result at bit 16, (Perform no rounding) extract the high 16 bits to produce a 1.15 result. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac32CmplxWithMvN Example

```
r6 = (r1,r0) ; /* fractional signed complex multiply; place complex product in r6.h
(imaginary) and r6.l (real) */
r4 = (r7,r3*) (is) ; /* integer signed complex multiply; place complex product in r4.
h (imaginary) and r4.l (real) */
r0 = (r2*,r4*) ; /* fractional signed complex multiply; place complex product in r0.
h (imaginary) and r0.l (real) */
r6 = a1:0 = (r1,r0) ; /* fractional signed complex multiply; place complex product in
a1 (imaginary) and a0 (real); move to r6 */
r4 = a1:0 = (r7,r3*) (is) ; /* integer signed complex multiply; place complex product
in a1 (imaginary) and a0 (real); move to r4 */
r0 = a1:0 = (r2*,r4*) ; /* fractional signed complex multiply; place complex product
in a1 (imaginary) and a0 (real); move to r0 */
r4 = a1:0 += (r3*,r1*) (is) ;  /* integer signed complex mac; accumulate complex result
in a1 (imaginary) and a0 (real); move to r4 */
r0 = a1:0 += (r5,r2*) ; /* fractional signed complex mac; accumulate complex result
in a1 (imaginary) and a0 (real); move to r0 */
r2 = a1:0 += (r6,r1) ; /* fractional signed complex mac; accumulate complex result in
a1 (imaginary) and a0 (real); move to r2 */
r6 = a1:0 -= (r2,1) (is) ;  /* integer signed complex mac; accumulate complex result
in a1 (imaginary) and a0 (real); move to r6 */
r0 = a1:0 -= (r3*,r7*) /* fractional signed complex mac; accumulate complex result in
a1 (imaginary) and a0 (real); move to r0 */
r2 = a1:0 -= (r4,r5*) (is) ; /* integer signed complex mac; accumulate complex result
in a1 (imaginary) and a0 (real); move to r2 */
```

16 x 16-Bit Multiply (Mult16)

General Form

Dsp32Mult
DREG_L = MUL0 MMOD1
DREG_H = MUL1 MMLMMOD1
DREG_E = MUL0 MMODE
DREG_O = MUL1 MMLMMODE

Abstract

This instruction multiplies two 16-bit half word operands. It stores, adds, or subtracts the product into a designated accumulator register with saturation.

See Also (MultInt, Mult32)

Mult16 Description

The multiply 16-bit operands instruction multiplies the two 16-bit operands and stores the result directly into the destination register with saturation.

NOTE: This instruction is similar to the multiply-accumulate instructions, *except that* the multiply 16-bit operands does not affect the accumulators.

Operations performed by the multiply-and-accumulate unit 0 (MAC0) portion of the architecture load their 16-bit results into the lower half of the destination data register; 32-bit results go into an even numbered data register. Operations performed by MAC1 load their results into the upper half of the destination data register or an odd numbered data register.

In **32-bit result syntax** (result goes to a 32-bit data register), the MAC performing the operation is determined by the destination data register. Instructions placing results in even-numbered data registers (R6, R4, R2, or R0) execute on MAC0 and may use *MMODE* options. Instructions placing results in odd-numbered data registers (R7, R5, R3, or R1) execute on MAC1 and may use *MMLMMODE* options. For example, 32-bit result operations with the (m) option may only be performed using odd-numbered data register destinations.

In **16-bit result syntax** (result goes to a 16-bit half data register), the MAC performing the operation is determined by the destination data register half. Instructions placing results in low-half data registers (R7.L through R0.L) execute on MAC0 and may use *MMOD1* options. Instructions placing results in high-half data registers (R7.H through R0.H) execute on MAC1 and may use *MMLMMOD1* options. For example, 16-bit result operations using the (m) option may only be performed using high-half data register destinations.

The versions of this instruction that produce 16-bit results are affected by the RND_MOD bit in the ASTAT register when they copy the results into the 16-bit destination register. RND_MOD determines whether biased or unbiased rounding is used. RND_MOD controls rounding for all versions of this instruction that produce 16-bit results except the (is), (iu) and (iss2) options.

In the syntax, where *MMOD1* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (ih), (is), (iss2), (iu), (s2rnd), (t), or (tfu).

In the syntax, where *MMLMMOD1* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (ih), (is), (iss2), (iu), (m), (m,fu), (m,ih), (m,is), (m,iss2), (m,iu), (m,s2rnd), (m,t), (m,tfu), (s2rnd), (t), or (tfu).

In the syntax, where *MMODE* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (is), (iss2), or (s2rnd).

In the syntax, where *MMLMMODE* appears, substitute a MAC mode for the accumulator copy format option: default (none), (fu), (is), (iss2), (m), (m,fu), (m,is), (m,iss2), (m,s2rnd), or (s2rnd).

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mult16 Example

```
r3.l = r3.h * r2.h ; /* MAC0. Both operands are signed fractions. */
r3.h = r6.h * r4.l (fu) ; /* MAC1. Both operands are unsigned fractions. */
r6 = r3.h * r4.h ; /* MAC0. Signed fraction operands, results saved as 32 bits. */
```

32 x 32-bit Multiply (Mult32)

General Form

`Dsp32Mult`
`DREG = DREG * DREG M32MMOD2`
`DREG_PAIR = DREG * DREG M32MMOD`

Abstract

This instruction executes multiply operations on 32-bit registers and on register pairs.

See Also (`MultInt`, `Mult16`)

Mult32 Description

The multiply 32-bit operands instruction multiplies two 32-bit half-word operands. It stores the product into a designated data register or data register pair with saturation.

In the syntax, where *`M32MMOD2`* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(is,ns)`, `(iu)`, `(iu,ns)`, `(m)`, `(m,is)`, `(m,is,ns)`, `(m,t)`, `(t)`, or `(tfu)`.

In the syntax, where *`MM32MMOD`* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(is,ns)`, `(iu)`, `(iu,ns)`, `(m)`, `(m,is)`, or `(m,is,ns)`.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mult32 Example

```
r0 = r0 * r3 (fu) ; /* fractional unsigned, place product in r0 */
r1:0 = r4 * r7 (fu) ; /* fractional unsigned, place product in r1:0 register pair */
```

32 x 32-Bit Multiply, Integer (MultInt)

General Form

ALU2op
DREG *= DREG

Abstract

This instruction does a //C style//, modulo 32-bit multiply with no saturation.

See Also (Mult16, Mult32)

MultInt Description

The multiply 32-Bit operands instruction multiplies two 32-bit data registers (dest_reg and multiplier_ register) and saves the product in *dest_reg*. The instruction mimics multiplication in the C language and effectively performs *Dreg1* = (*Dreg1* * *Dreg2*) modulo 2^{32}. Since the integer multiply is modulo 2^{32}, the result always fits in a 32-bit *dest_reg*, and overflows are possible but not detected. The overflow status bit in the ASTAT register is never set.

Users are required to limit input numbers to ensure that the resulting product does not exceed the 32-bit *dest_reg* capacity. If overflow notification is required, users should write their own multiplication macro with that capability.

Accumulators A0 and A1 are unchanged by this instruction.

The multiply 32-bit operands instruction does not implicitly modify the number in multiplier_register.

This instruction might be used to implement the congruence method of random number generation according to:

Figure 8-6: Integer Multiply Equation

$$X[n+a] = (a \times X[n]) \bmod 2^{32}$$

where:

- X[n] is the seed value,
- a is a large integer, and
- X[n+1] is the result that can be multiplied again to further the pseudo-random sequence.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

MultInt Example

```
r3 *= r0 ; /* equivalent to r3 = r3 * r0 */
```

Dual 16 x 16-Bit MAC (ParaMac16AndMac16)

General Form

Dsp32Mac
MAC1 MML , MAC0 MMOD0
MAC1 MML , MAC0 MMOD0
MAC1 MML , MAC0 MMOD0
MAC1 MML , MAC0 MMOD0

Abstract

This dual multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second MAC operation occurs in parallel.

See Also (ParaMac16WithMvAndMac16WithMv, ParaMac16AndMac16WithMv, ParaMac16WithMvAndMac16)

ParaMac16AndMac16 Description

The dual multiply and multiply-accumulate to accumulator instruction is a dual (two instances issued in parallel) of the 16 x 16-Bit MAC (Mac16) instruction. For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16AndMac16 Example

```
a1 += r3.h * r2.h , a0 = r3.h * r2.h ;/
a1 -= r6.h * r4.l (fu) , a0 += r6.h * r4.l (fu) ;
a1 = r6.h * r4.l (fu) , a0 -= r3.h * r2.h ;
```

Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMac16WithMv)

General Form

Dsp32Mac
MAC1 MML , DREG_L =(MAC0) MMOD1
MAC1 MML , DREG_L =(MAC0) MMOD1
MAC1 MML , DREG_L =(MAC0) MMOD1
MAC1 MML , DREG_L =(MAC0) MMOD1
MAC1 MML , DREG_E =(MAC0) MMODE
MAC1 MML , DREG_E =(MAC0) MMODE
MAC1 MML , DREG_E =(MAC0) MMODE
MAC1 MML , DREG_E =(MAC0) MMODE

Abstract

This dual multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second MAC operation occurs in parallel.

See Also (ParaMac16AndMac16, ParaMac16WithMvAndMac16WithMv, ParaMac16WithMvAndMac16)

ParaMac16AndMac16WithMv Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with an instance of the the 16 x 16-Bit MAC (Mac16) instruction (using MAC1) and an instance of the 16 x 16-Bit MAC with Move to Register (Mac16WithMv) instruction (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16AndMac16WithMv Example

```
a1 += r6.h * r4.l (fu) , r3.l = ( a0 = r3.h * r2.h ) ;
```

Dual 16 x 16-Bit MAC with Move to Register (Para-Mac16WithMvAndMac16)

General Form

Dsp32Mac
DREG_H =(MAC1) MML , MAC0 MMOD1
DREG_H =(MAC1) MML , MAC0 MMOD1
DREG_H =(MAC1) MML , MAC0 MMOD1
DREG_H =(MAC1) MML , MAC0 MMOD1
DREG_O =(MAC1) MML , MAC0 MMODE
DREG_O =(MAC1) MML , MAC0 MMODE
DREG_O =(MAC1) MML , MAC0 MMODE
DREG_O =(MAC1) MML , MAC0 MMODE

Abstract

This dual multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second MAC operation occurs in parallel.

See Also (ParaMac16AndMac16, ParaMac16WithMvAndMac16WithMv, ParaMac16AndMac16WithMv)

ParaMac16WithMvAndMac16 Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with an instance of the 16 x 16-Bit MAC with Move to Register (Mac16WithMv) instruction (using MAC1) and an instance of the 16 x 16-Bit MAC (Mac16) (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16WithMvAndMac16 Example

```
r3.h = (a1 += r6.h * r4.l) (fu) , a0 = r3.h * r2.h ;
```

Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMac16WithMv)

General Form

Dsp32Mac
DREG_H =(MAC1) MML , DREG_L =(MAC0) MMOD1
DREG_H =(MAC1) MML , DREG_L =(MAC0) MMOD1
DREG_H =(MAC1) MML , DREG_L =(MAC0) MMOD1
DREG_H =(MAC1) MML , DREG_L =(MAC0) MMOD1
DREG_O =(MAC1) MML , DREG_E =(MAC0) MMODE
DREG_O =(MAC1) MML , DREG_E =(MAC0) MMODE
DREG_O =(MAC1) MML , DREG_E =(MAC0) MMODE
DREG_O =(MAC1) MML , DREG_E =(MAC0) MMODE

Abstract

This dual multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second MAC operation occurs in parallel.

See Also (ParaMac16AndMac16, ParaMac16AndMac16WithMv, ParaMac16WithMvAndMac16)

ParaMac16WithMvAndMac16WithMv Description

The dual multiply and multiply-accumulate to accumulator (with move) instruction is a dual (two instances issued in parallel) of the 16 x 16-Bit MAC with Move to Register (Mac16WithMv) instruction. For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation input operands. The instructions must NOT use the same data registers for results.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16WithMvAndMac16WithMv Example

```
r1 = (a1 = r3 * r0) (m,fu) , r0 = (a0 = r0 * r3) (fu) ;
r3 = (a1 += r4 * r7) (m,is) , r2 = (a0 += r1 * r2) (is) ;
r1 = (a1 = r3 * r0) (m,fu) , r4 = (a0 -= r2 * r1) (iss2) ;
```

Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMv)

General Form

Dsp32Mac
MAC1 MML , DREG_L = a0 MMOD1
MAC1 MML , DREG_L = a0 MMOD1
MAC1 MML , DREG_E = a0 MMODE
MAC1 MML , DREG_E = a0 MMODE

Abstract

This dual move and multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second (independent) move operation occurs in parallel with the MAC operation.

See Also (ParaMac16WithMvAndMv)

ParaMac16AndMv Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with an instance of the the 16 x 16-Bit MAC (Mac16) instruction (using MAC1) and *either* an instance of the Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL) instruction (using MAC0) *or* an instance of the Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE) instruction (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	**AV1S**	**AV1**	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16AndMv Example

```
a1 += r6.h * r4.l (fu) , r3.l = a0 ;
a1 += r6.h * r4.l (fu) , r2 = a0 ;
```

Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMv)

General Form

Dsp32Mac
DREG_H =(MAC1) MML , DREG_L =a0 MMOD1
DREG_H =(MAC1) MML , DREG_L =a0 MMOD1
DREG_O =(MAC1) MML , DREG_E =a0 MMODE
DREG_O =(MAC1) MML , DREG_E =a0 MMODE

Abstract

This dual move and multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second (independent) move operation occurs in parallel with the MAC operation.

See Also (ParaMac16AndMv)

ParaMac16WithMvAndMv Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with an instance of the 16 x 16-Bit MAC with Move to Register (Mac16WithMv) instruction (using MAC1) and *either* an instance of the Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL) instruction (using MAC0) *or* an instance of the Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE) instruction (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	**AV1S**	**AV1**	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16WithMvAndMv Example

```
r1.h = (a1 += r6.h * r4.l) (fu) , r3.l = a0 ;
r1.h = (a1 += r6.h * r4.l) (fu) , r2 = a0 ;
r2 = (a1 += r6.h * r4.l) (fu) , r3.l = a0 ;
r0 = (a1 += r6.h * r4.l) (fu) , r2 = a0 ;
```

Dual 16 x 16-Bit Multiply (ParaMult16AndMult16)

General Form

```
Dsp32Mult
DREG_H = MUL1 MML , DREG_L = MUL0 MMOD1
```

Abstract

This instruction executes a two parallel multiply operations on 16-bit registers.

ParaMult16AndMult16 Description

The dual multiply 16-bit operands instruction is a dual (two instances issued in parallel) of the 16 x 16-Bit Multiply (Mult16) instruction. One of the parallel issue instructions executes on MAC1 with its results placed in a high half data register. The other parallel issue instruction executes on MAC0 with its results placed in a low half data register. For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMult16AndMult16 Example

```
r3.h = r6.h * r4.l (fu) , r3.l = r3.h * r2.h ;
```

Dual Move to Register and 16 x 16-Bit MAC (ParaMvAndMac16)

General Form

Dsp32Mac
DREG_H = a1 MML , MAC0 MMOD1
DREG_H = a1 MML , MAC0 MMOD1
DREG_O = a1 MML , MAC0 MMODE
DREG_O = a1 MML , MAC0 MMODE

Abstract

This dual move and multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second (independent) move operation occurs in parallel with the MAC operation.

See Also (ParaMvAndMac16WithMv)

ParaMvAndMac16 Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with *either* an instance of the Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH) instruction (using MAC1) *or* an instance of the Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO) instruction (using MAC1) and an instance of the 16 x 16-Bit MAC (Mac16) instruction (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMvAndMac16 Example

```
r3.l = a1 , a0 += r6.h * r4.l (fu) ;
r2 = a1 , a0 += r6.h * r4.l (fu) ;
```

Dual Move to Register and 16 x 16-Bit MAC with Move to Register (ParaMvAndMac16WithMv)

General Form

Dsp32Mac
DREG_H =a1 MML , DREG_L =(MAC0) MMOD1
DREG_H =a1 MML , DREG_L =(MAC0) MMOD1
DREG_O =a1 MML , DREG_E =(MAC0) MMODE
DREG_O =a1 MML , DREG_E =(MAC0) MMODE

Abstract

This dual move and multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second (independent) move operation occurs in parallel with the MAC operation.

See Also (ParaMvAndMac16)

ParaMvAndMac16WithMv Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with *either* an instance of the Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH) instruction (using MAC1) *or* an instance of the Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO) instruction (using MAC1) and an instance of the 16 x 16-Bit MAC with Move to Register (Mac16WithMv) instruction (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMvAndMac16WithMv Example

```
r3.h = a1 , r1.h = (a0 += r6.h * r4.l) (fu) ;
r3 = a1 , r1.h = (a0 += r6.h * r4.l) (fu) ;
r3.h = a1 , r2 = (a0 += r6.h * r4.l) (fu) ;
r7 = a1 , r0 = (a0 += r6.h * r4.l) (fu) ;
```

Pointer Math Operations

These operations provide addition and/or subtract operations on pointer register and immediate value operands:

- 32-bit Add or Subtract (DagAdd32)
- 32-bit Add then Shift (DagAddSubShift)
- 32-bit Add or Subtract Constant (DagAddImm)
- 32-bit Add Shifted Pointer (PtrOp)
- Pointer Logical Shift (LShiftPtr)

32-bit Add or Subtract (DagAdd32)

General Form

`Ptr2op`
`PREG -= PREG`
`PREG += PREG` (brev)
`Comp3op`
`PREG = PREG + PREG`
`DAGModIm`
`IREG += MREG`
`IREG += MREG` (brev)
`IREG -= MREG`

Abstract

This instruction adds or subtracts two pointer registers.

See Also (`DagAddSubShift`, `DagAddImm`, `PtrOp`)

DagAdd32 Description

The DAG AddSub32 instruction adds or subtracts source pointer registers and places the result in a destination pointer register.

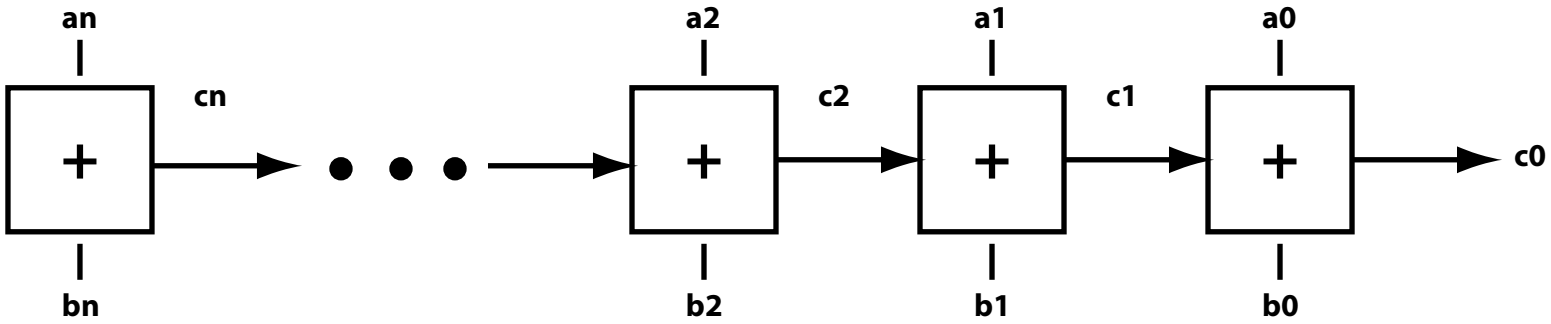
The instruction versions that explicitly modify an index register (*Ireg*) support optional circular buffering. For more information, see Addressing Circular Buffers in the Address Arithmetic Unit (AAU) chapter. Unless circular buffering is desired, disable this feature prior to issuing this instruction by clearing the length register (*Lreg*) corresponding to the *Ireg* used in this instruction. For example, if using the `i2` to increment your address pointer, first clear `l2` to disable circular buffering. Failure to explicitly clear the corresponding *Lreg* beforehand can result in unexpected *Ireg* values.

The circular address buffer registers (index, length, and base) are not initialized automatically by processor reset. The recommended operation is that user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes these registers later, if needed.

When the bit reverse carry adder (BREV) is specified in the instruction syntax, the carry bit is *propagated from left-to-right*, as shown in the **Bit Addition Flow for the Bit Reverse (BREV) Case** figure, instead of being *propagated from right-to-left* (default operation). When bit reversal is used on the index register

version of this instruction, circular buffering is disabled to support operand addressing for FFT, DCT, and DFT algorithms. The pointer register version of this instruction does not support circular buffering.

Figure 8-7: Bit Addition Flow for the Bit Reverse (BREV) Case



This instruction has a *special application*, regarding load or store operations. Typically, programs use the index register and pointer register versions of this instruction to increment or decrement indirect address pointers for load or store operations.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

DagAdd32 Example

```
p5 = p3 + p0 ; /* dest_Preg = src1_Preg + src0_Preg */
p3 -= p0 ; /* dest_Preg_new = dest_Preg_old - src_Preg */
i1 -= m2 ; /* dest_Ireg_new = dest_Ireg_old - src_Mreg */
p3 += p0 (brev) ; /* dest_Preg_new = dest_Preg_old + src_Preg (bit reversed carry,
only) */
i1 += m1 ; /* dest_Ireg_new = dest_Ireg_old + src_Mreg */
i0 += m0 (brev) ; /* optional bit reverse carry, only */
```

32-bit Add or Subtract Constant (DagAddImm)

General Form

CompI2opP
PREG += imm7
DAGModIk
IREG += 2
IREG -= 2
IREG += 4
IREG -= 4

Abstract

This instruction allows the user to add a constant to a register.

See Also (DagAdd32, DagAddSubShift, PtrOp)

DagAddImm Description

The DAG AddImm instruction adds or subtracts a source pointer registers and a constant value, then places the result in a destination pointer register.

The instruction versions that explicitly modify an index register (*Ireg*) support optional circular buffering. For more information, see Addressing Circular Buffers in the Address Arithmetic Unit (AAU) chapter. Unless circular buffering is desired, disable this feature prior to issuing this instruction by clearing the length register (*Lreg*) corresponding to the *Ireg* used in this instruction. For example, if using the i2 to increment your address pointer, first clear l2 to disable circular buffering. Failure to explicitly clear the corresponding *Lreg* beforehand can result in unexpected *Ireg* values.

The circular address buffer registers (index, length, and base) are not initialized automatically by processor reset. The recommended operation is that user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes these registers later, if needed

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

DagAddImm Example

```
p5 += -8 ; /* Preg = Preg + constant */
i0 += 2 ;  /* Ireg = Ireg + 2 */
i1 += 4 ;  /* Ireg = Ireg + 4 */
```

```
i2 -= 2 ;  /* Ireg = Ireg - 2 */
i0 -= 4 ;  /* Ireg = Ireg - 4 */
```

32-bit Add then Shift (DagAddSubShift)

General Form

Ptr2op
PREG =(PREG + PREG)<< 1
PREG =(PREG + PREG)<< 2

Abstract

This instruction adds then shift left one or two places. Saturation is not supported.

See Also (DagAdd32, DagAddImm, PtrOp)

DagAddSubShift Description

The add with shift instruction adds two source pointer register, then applies a one- or two-bit logical shift left. The left shift accomplishes a x2 or x4 multiplication on sign-extended numbers.

The add with shift instruction does not intrinsically modify values that are strictly input. However, the *dest_reg* serves as an input as well as the result, so the *dest_reg* is intrinsically modified.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

DagAddSubShift Example

```
p3 = (p3 + p2) << 1 ;
/* dest_reg = (dest_reg + src_reg) x 2 */
/* p3 = (p3 + p2) * 2 */

p3 = (p3 + p2) << 2 ;
/* dest_reg = (dest_reg + src_reg) x 4 (a) */
/* p3 = (p3 + p2) * 4 */
```

32-bit Add Shifted Pointer (PtrOp)

General Form

Comp3op
PREG = PREG + (PREG << 1)
PREG = PREG + (PREG << 2)

Abstract

This instruction adds or subtracts pointer and DAG registers.

See Also (DagAdd32, DagAddSubShift, DagAddImm)

PtrOp Description

The shift with add instruction combines a one- or two-bit logical shift left with an addition operation.

The instruction provides a shift-then-add method that supports a rudimentary multiplier sequence useful for array pointer manipulation.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

PtrOp Example

```
p3 = p0 + (p3 << 1) ;
/* p3 = (p3 * 2) + p0 */
/* adder_pntr + (src_pntr * 2) */

p3 = p0 + (p3 << 2) ;
/* p3 = (p3 * 4) + p0 */
/* adder_pntr + (src_pntr * 4) */
```

Pointer Logical Shift (LShiftPtr)

General Form

Ptr2op
PREG = PREG << 2
PREG = PREG << 1
PREG = PREG >> 2
PREG = PREG >> 1

Abstract

This instruction shifts a pointer register by the specified number of bits.

LShiftPtr Description

The logical shift pointer instruction logically shifts a pointer register by a specified distance and direction.

Logical shifts discard any bits shifted out of the register and backfill vacated bits with zeros.

The logical shift pointer instruction does not implicitly modify the input *src_pntr* value. However, the dest_pntr can be the same pointer register as *src_pntr*. Doing so explicitly modifies the source register.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

LShiftPtr Example

```
p3 = p2 >> 1 ; /* pointer right shift by 1 */
p3 = p3 >> 2 ; /* pointer right shift by 2 */
p4 = p5 << 1 ; /* pointer left shift by 1 */
p0 = p1 << 2 ; /* pointer left shift by 2 */
```

Rotate Operations

These operations provide bitwise rotate operations on register and immediate value operands:

- 32-Bit Rotate (Shift_Rot32)
- Accumulator Rotate (Shift_RotAcc)

32-Bit Rotate (Shift_Rot32)

General Form

Dsp32Shf
DREG = rot DREG by DREG_L
Dsp32ShfImm
DREG = rot DREG by imm6

Abstract

This instruction rotates the a register through the CC bit a specified distance and direction. The CC bit is in the rotate chain.

Shift_Rot32 Description

The rotate data register instruction rotates a data register through the CC bit a specified distance and direction. The CC bit is in the rotate chain. Consequently, the first value rotated into the register is the initial value of the CC bit.

Rotation shifts all the bits either right or left. Each bit that rotates out of the register (the LSB for rotate right or the MSB for rotate left) is stored in the CC bit, and the CC bit is stored into the bit vacated by the rotate on the opposite end of the register.

Figure 8-8: Left-versus-Right Bit Rotation Example

```
If                          31                                      0
D-register:                 1010 1111 0000 0000 0000 0000 0001 1010
CC bit:                     N (“1” or “0”)

Rotate left 1 bit           31                                      0
D-register:                 0101 1110 0000 0000 0000 0000 0011 010N
CC bit:                     1

Rotate left 1 bit again     31                                      0
D-register:                 1011 1100 0000 0000 0000 0000 0110 10N1
CC bit: 0

If                          31                                      0
D-register:                 1010 1111 0000 0000 0000 0000 0001 1010
CC bit:                     N (“1” or “0”)

Rotate right 1 bit          31                                      0
D-register:                 N101 0111 1000 0000 0000 0000 0000 1101
CC bit:                     0

Rotate right 1 bit again    31                                      0
D-register:                 0N10 1011 1100 0000 0000 0000 0000 0110
CC bit:                     1
```

The sign of the rotate magnitude determines the direction of the rotation.

- Positive rotate magnitudes produce Left rotations.
- Negative rotate magnitudes produce Right rotations.

Valid rotate magnitudes are –32 through +31, zero included. The rotate instruction masks and ignores bits that are more significant than those allowed. The distance is determined by the lower 6 bits (sign extended) of the *shift_magnitude*.

Unlike shift operations, the rotate instruction loses no bits of the source register data. Instead, it rearranges them in a circular fashion. However, the last bit rotated out of the register remains in the CC bit, and is not returned to the register. Because rotates are performed all at once and not one bit at a time, rotating one direction or another regardless of the rotate magnitude produces no advantage. For instance, a rotate right by two bits is no more efficient than a rotate left by 30 bits. Both methods produce identical results in identical execution time.

This instruction rotates all 32 bits of the data register.

The instruction does not implicitly modify the *src_reg* values. Optionally, *dest_reg* can be the same data register as *src_reg*. Doing this explicitly modifies the source register.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	**CC**	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift_Rot32 Example

```
r4 = rot r1 by 8 ; /* rotate left (Dreg = ROT Dreg BY imm6) */
r4 = rot r1 by -5 ; /* rotate right */
```

Accumulator Rotate (Shift_RotAcc)

General Form

Dsp32Shf
a0 = rot a0 by DREG_L
a1 = rot a1 by DREG_L
Dsp32ShfImm
a0 = rot a0 by imm6
a1 = rot a1 by imm6

Abstract

This instruction rotates the accumulator through the CC bit a specified distance and direction. The CC bit is in the rotate chain.

Shift_RotAcc Description

This instruction rotates the accumulator through the CC bit a specified distance and direction. The CC bit is in the rotate chain. Consequently, the first value rotated into the register is the initial value of the CC bit and the last bit rotated out ends up in CC. The sign of the rotate magnitude determines the direction of the rotation.

- Positive rotates left
- Negative rotates right

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	**CC**	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift_RotAcc Example

```
a0 = rot a0 by 22 ; /* rotate Accumulator left */
a1 = rot a1 by -31 ; /* rotate Accumulator right */
a0 = rot a0 by r3.l ;
a1 = rot a1 by r7.l ;
```

Shift Operations

These operations provide arithmetic or logical shift operations on register and immediate value operands:

- 16-Bit Arithmetic Shift (AShift16)
- Vectored 16-Bit Arithmetic (AShift16Vec)
- Accumulator Arithmetic Shift (AShiftAcc)
- 32-Bit Arithmetic Shift (AShift32)
- 16-Bit Logical Shift (LShift16)
- Vectored 16-Bit Logical Shift (LShift16Vec)
- 32-Bit Logical Shift (LShift)
- Accumulator Logical Shift (LShiftA)

16-Bit Arithmetic Shift (AShift16)

General Form

Dsp32Shf
DREG_L = ashift DREG_L by DREG_L
DREG_L = ashift DREG_H by DREG_L
DREG_H = ashift DREG_L by DREG_L
DREG_H = ashift DREG_H by DREG_L
DREG_L = ashift DREG_L by DREG_L (s)
DREG_L = ashift DREG_H by DREG_L (s)
DREG_H = ashift DREG_L by DREG_L (s)
DREG_H = ashift DREG_H by DREG_L (s)
Dsp32ShfImm
DREG_L = DREG_L AHSH4
DREG_L = DREG_H AHSH4
DREG_H = DREG_L AHSH4
DREG_H = DREG_H AHSH4
DREG_L = DREG_L AHSH4S
DREG_L = DREG_H AHSH4S
DREG_H = DREG_L AHSH4S
DREG_H = DREG_H AHSH4S

Abstract

This instruction shifts left or right and preserves the sign bit. For right shifts, the sign bit back-fills the left-most bit vacated by the shift. For left shifts, if the shift causes the sign bit to be lost, the result will saturate to the maximum positive or negative value depending on the lost sign bit.

See Also (AShift16Vec)

AShift16 Description

The arithmetic shift low/high half data register destination instruction shifts a register contents a specified distance (*shift_magnitude*) and direction (based on syntax and/or *shift_magnitude* sign) while preserving the sign bit of the original number. This instruction provides arithmetic shift right, logical shift left, and arithmetic shift left (with saturation) operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on **The Science Dictionary** site are helpful.

The versions of this instruction using `ashift` syntax support the following shift operations:

- For a positive *`shift_magnitude`*, `ashift` produces an arithmetic shift right with sign bit preservation. The sign bit value back-fills the left-most bit positions vacated by the arithmetic shift right.
- For a negative *`shift_magnitude`*, `ashift` produces a logical shift left, but does not guarantee sign bit preservation. If the negative *`shift_magnitude`* is too large, the `ashift` operation saturates the destination register. A logical shift left that would otherwise lose non-sign bits off the left-hand side saturates to the maximum positive or negative value instead.

NOTE: One may view the `ashift` operation as a multiplication or division operation. Viewed this way, the *`shift_magnitude`* is the power of 2 multiplied by the *`src_reg`* number. Positive magnitudes cause multiplication ($N \times 2^n$), and negative magnitudes produce division ($N \times 2^{-n}$ or $N / 2^n$).

The versions of this instruction using >>> syntax only support arithmetic right shift operations using positive *`shift_magnitude`* values.

The versions of this instruction using <<< syntax only support logical left shift operations using positive *`shift_magnitude`* values.

The versions of this instruction using << with `(s)` syntax only support arithmetic shift left operations (with saturation) using positive *`shift_magnitude`* values.

The **Arithmetic Shift (16 Bit Destination Register) Operations** table provide more detailed information about arithmetic shift operations.

Table 8-12: Arithmetic Shift (16 Bit Destination Register) Operations

Table 8-13:

Syntax	Description
">>>", << (with saturation), and `ashift`	The value in *`src_reg`* is shifted by the number of places specified in *`shift_magnitude`*, and the result is stored into *`dest_reg`*. The `ashift` versions can shift 16-bit Dreg_lo_hi registers by up to –16 through +15 places.

The *`dest_reg`* and *`src_reg`* may be a 16-bit half data register.

For 16-bit *`src_reg`*, valid shift magnitudes are –16 through +15, zero included.

The data register versions of this instruction shift 16 bits for half-word registers.

The half data register versions of this instruction do not implicitly modify the *`src_reg`* values. Optionally, *`dest_reg`* may be the same data register as *`src_reg`*. Doing this explicitly modifies the source register.

Where permitted (optional) or required the saturation (s) option applies saturation of the result. For shift operations *without saturation enabled*, values may be left-shifted so far that all the sign bits overflow and are lost. For shift operations *with saturation enabled*, a left shift that would otherwise shift nonsign bits off the left-hand side saturates to the maximum positive or negative value instead. The result always keeps the same sign as the pre-shifted value when saturation is enabled.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instructions** can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AShift16 Example

```
/* AShift16 syntax summary */
/* Dreg_lo_hi = ashift Dreg_lo_hi BY Dreg_lo (optional_sat) ; arithmetic or logical
shift with optional saturation*/
/* Dreg_lo_hi = Dreg_lo_hi >>> uimm4 (optional_sat) ; arithmetic shift right with
optional saturation*/
/* Dreg_lo_hi = Dreg_lo_hi <<< uimm4 ; logical shift left*/
/* Dreg_lo_hi = Dreg_lo_hi <<< 0 ; logical shift left*/
/* Dreg_lo_hi = Dreg_lo_hi << uimm4 (s) ; arithmetic shift left with saturation*/
/* Dreg_lo_hi = Dreg_lo_hi << 0 (s) ; arithmetic shift left with saturation*/
/* AShift16 syntax examples */
r3.l = r0.h >>> 7 ; /* arithmetic right shift, half-word */
r3.h = r0.h >>> 5 ; /* same as above; any combination of upper and lower half-words
is supported */
r3.l = r0.h >>> 7(s) ; /* arithmetic right shift, half-word, saturated */
r3.l = r0.h << 12 (s) ; /* arithmetic left shift */
r3.l = ashift r0.h by r7.l ; /* shift, half-word */
r3.h = ashift r0.l by r7.l ;
r3.h = ashift r0.h by r7.l ;
r3.l = ashift r0.l by r7.l ;
r3.l = ashift r0.h by r7.l(s) ; /* shift, half-word, saturated */
r3.h = ashift r0.l by r7.l(s) ; /* shift, half-word, saturated */
r3.h = ashift r0.h by r7.l(s) ;
r3.l = ashift r0.l by r7.l (s) ;
/* If r0.h = -64, then performing . . . */
r3.h = r0.h >>> 4 ;
/* . . . produces r3.h = -4, preserving the sign */
```

Vectored 16-Bit Arithmetic (AShift16Vec)

General Form

Dsp32Shf
DREG = ashift DREG by DREG_L (v)
DREG = ashift DREG by DREG_L (v,s)
Dsp32ShfImm
DREG = DREG AHSH4 (v)
DREG = DREG AHSH4VS

Abstract

This instruction shifts a 16-bit vector left or right by the value in the XOP register. When shifting right, the sign bit will be replicated. If saturation is specified, ASHIFT lefts will saturate if any of the bits shifted off do not match the original sign bit.

See Also (AShift16)

AShift16Vec Description

The arithmetic shift data register destination (vector) instruction performs two independent shifts, shifting a contents of a register's low half and high half a specified distance (*shift_magnitude*) and direction (based on syntax and/or *shift_magnitude* sign) while preserving the sign bits of the original numbers. Although the two half-word registers are shifted at the same time, the two numbers are kept separate. This instruction provides arithmetic shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on **The Science Dictionary** site are helpful.

The versions of this instruction using ashift syntax support the following shift operations:

- For a positive *shift_magnitude*, ashift produces an arithmetic shift right with sign bit preservation. The sign bit value back-fills the left-most bit positions vacated by the arithmetic shift right.
- For a negative *shift_magnitude*, ashift produces a logical shift left, but does not guarantee sign bit preservation. If the negative *shift_magnitude* is too large, the ashift operation saturates the destination register. A logical shift left that would otherwise lose non-sign bits off the left-hand side saturates to the maximum positive or negative value instead.

NOTE: One may view the ashift operation as a multiplication or division operation. Viewed this way, the *shift_magnitude* is the power of 2 multiplied by the *src_reg* number. Positive magnitudes cause multiplication ($N \times 2^n$), and negative magnitudes produce division ($N \times 2^{-n}$ or $N / 2^n$).

The versions of this instruction using >>> syntax only support arithmetic right shift operations using positive *shift_magnitude* values.

The versions of this instruction using <<< syntax only support logical left shift operations using positive *shift_magnitude* values.

The versions of this instruction using << with (v,s) syntax only support logical shift left operations (with saturation) using positive *shift_magnitude* values.

The **Arithmetic Shift (16 Bit Destination Register) Operations** table provide more detailed information about arithmetic shift operations.

Table 8-14: Arithmetic Shift (16 Bit Destination Register) Operations

Table 8-15:

Syntax	Description
">>>", << (with saturation), and ashift	The value in *src_reg* is shifted by the number of places specified in *shift_magnitude*, and the result is stored into *dest_reg*. The ashift versions can shift the two 16-bit halves of a Dreg (independently) by up to –16 through +15 places.

The *dest_reg* and *src_reg* may be a 132-bit register.

For each 16-bit half of the *src_reg*, valid shift magnitudes are –16 through +15, zero included.

The data register versions of this instruction shift 16 bits for the two half-word sections of the word registers, independently.

The data register versions of this instruction always modify the *src_reg* values.

Where permitted (optional) or required the saturation (s) option applies saturation of the result. For shift operations *without saturation enabled*, values may be left-shifted so far that all the sign bits overflow and are lost. For shift operations *with saturation enabled*, a left shift that would otherwise shift nonsign bits off the left-hand side saturates to the maximum positive or negative value instead. The result always keeps the same sign as the pre-shifted value when saturation is enabled.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AShift16Vec Example

```
/* AShift16Vec syntax summary */
Dreg1 = ashift Dreg1 by Dreg0_lo (v) /* arithmetic/logical shift, vector (dual) */
Dreg1 = ashift Dreg1 by Dreg0_lo (v,s) /* arithmetic/logical shift, vector (dual) */
Dreg = Dreg <<< 0 (v) /* logical shift left, vector (dual) */
Dreg = Dreg <<< UImm4 (v) /* logical shift left, vector (dual) */
Dreg = Dreg >>> UImm4N (v) /* arithmetic shift right, vector (dual) */
Dreg = Dreg << 0 (v,s) /* logical shift left with saturation, vector (dual) */
Dreg = Dreg << UImm4 (v,s) /* logical shift left with saturation, vector (dual) */
Dreg = Dreg >>> UImm4N (v,s) /* arithmetic shift right with saturation, vector (dual)
*/
/* AShift16Vec syntax examples */
r4=r5>>3 (v) ; /* logical right shift immediate R5.H and R5.L by 3 bits */
r4=r5<<3 (v) ; /* logical left shift immediate R5.H and R5.L by 3 bits */
r2=lshift r7 by r5.l (v) ;
/* logically shift (right or left, depending on sign of r5.l) R7.H and R7.L by
magnitude of R5.L */
```

32-Bit Arithmetic Shift (AShift32)

General Form

ALU2op
DREG >>>= DREG
Logi2Op
DREG >>>= uimm5
Dsp32Shf
DREG = ashift DREG by DREG_L
DREG = ashift DREG by DREG_L (s)
Dsp32ShfImm
DREG = DREG ASH5
DREG = DREG ASH5S

Abstract

This instruction shifts left or right and preserves the sign bit. For right shifts, the sign bit back-fills the left-most bit vacated by the shift. For left shifts, if the shift causes the sign bit to be lost, the result will saturate to the maximum positive or negative value depending on the lost sign bit.

See Also (AShiftAcc)

AShift32 Description

The arithmetic shift data register destination instruction shifts a register contents a specified distance (*shift_magnitude*) and direction (based on syntax and/or *shift_magnitude* sign) while preserving the sign bit of the original number. This instruction provides arithmetic shift right, logical shift left, and arithmetic shift left (with saturation) operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on **The Science Dictionary** site are helpful.

The versions of this instruction using `ashift` syntax support the following shift operations:

- For a positive *shift_magnitude*, `ashift` produces an arithmetic shift right with sign bit preservation. The sign bit value back-fills the left-most bit positions vacated by the arithmetic shift right.
- For a negative *shift_magnitude*, `ashift` produces a logical shift left, but does not guarantee sign bit preservation. If the negative *shift_magnitude* is too large, the `ashift` operation saturates the destination register. A logical shift left that would otherwise lose non-sign bits off the left-hand side saturates to the maximum positive or negative value instead.

NOTE: One may view the ashift operation as a multiplication or division operation. Viewed this way, the *shift_magnitude* is the power of 2 multiplied by the *src_reg* number. Positive magnitudes cause multiplication (N x 2^n), and negative magnitudes produce division (N x 2^{-n} or N / 2^n).

The versions of this instruction using >>>= and >>> syntax only support arithmetic right shift operations using positive *shift_magnitude* values.

The versions of this instruction using <<< syntax only support logical left shift operations using positive *shift_magnitude* values.

The versions of this instruction using << with (s) syntax only support arithmetic shift left operations (with saturation) using positive *shift_magnitude* values.

The **Arithmetic Shift (32 Bit Destination Register) Operations** table provide more detailed information about arithmetic shift operations.

Table 8-16: Arithmetic Shift (32 Bit Destination Register) Operations

Table 8-17:

Syntax	Description
>>>=	The value in *dest_reg* is right-shifted by the number of places specified by *shift_magnitude*. The data size is always 32 bits long. The entire 32 bits of the *shift_magnitude* determine the shift value. Shift magnitudes larger than 0x1F result in either 0x00000000 (when the input value is positive) or 0xFFFFFFFF (when the input value is negative).
">>>", << (with saturation), and ashift	The value in *src_reg* is shifted by the number of places specified in *shift_magnitude*, and the result is stored into *dest_reg*. The ashift versions can shift 32-bit Dreg registers by up to –32 through +31 places.

The *dest_reg* and *src_reg* may be a 32-bit register.

For 32-bit *src_reg*, valid shift magnitudes are –32 through +31, zero included.

The data register versions of this instruction shift 32 bits for word registers.

The data register versions of this instruction do not implicitly modify the *src_reg* values. Optionally, *dest_reg* can be the same data register as *src_reg*. Doing this explicitly modifies the source register.

Where permitted (optional) or required the saturation (s) option applies saturation of the result. For shift operations *without saturation enabled*, values may be left-shifted so far that all the sign bits overflow and are lost. For shift operations *with saturation enabled*, a left shift that would otherwise shift nonsign bits off the left-hand side saturates to the maximum positive or negative value instead. The result always keeps the same sign as the pre-shifted value when saturation is enabled.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

The versions of this instruction using >>>, <<<, <<, and ashift syntax are **32-bit instructions**, which can sometimes save execution time (over a 16-bit encoded instruction) because they can be issued in parallel with certain other instructions.

The versions of this instruction using >>>= syntax are **16-bit instructions** (which takes up less memory space over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AShift32 Example

```
/* AShift32 syntax summary */
/* Dreg >>>= Dreg ; arithmetic shift right */
/* Dreg >>>= UImm5 ; arithmetic shift right */
/* Dreg = ashift Dreg by Dreg_lo (optional_sat) ; arithmetic/logical shift with
optional saturation */
/* Dreg = Dreg <<< 0 ; logical shift left */
/* Dreg = Dreg <<< UImm5 ; logical shift left */
/* Dreg = Dreg >>> UImm5N ; arithmetic shift right */
/* Dreg = Dreg << 0 (s) ; arithmetic shift left with saturation */
/* Dreg = Dreg << UImm5 (s) ; arithmetic shift left with saturation */
/* Dreg = Dreg >>> UImm5 ; arithmetic shift right */
/* AShift32 syntax examples */
r0 >>>= 19 ; /* 16-bit instruction length arithmetic right shift */
r0 >>>= r2 ; /* 16-bit instruction length arithmetic right shift */
r5 = r2 << 24 (s) ; /* arithmetic left shift */
r4 = ashift r2 by r7.l ; /* shift, word */
r4 = ashift r2 by r7.l (s) ; /* shift, word, saturated */
```

Accumulator Arithmetic Shift (AShiftAcc)

General Form

Dsp32Shf
a0 = ashift a0 by DREG_L
a1 = ashift a1 by DREG_L
Dsp32ShfImm
a0 = a0 ASH5
a1 = a1 ASH5

Abstract

This instruction shifts left or right and preserves the sign bit. For right shifts, the sign bit back-fills the left-most bit vacated by the shift.

See Also (AShift32)

AShiftAcc Description

The arithmetic shift accumulator register destination instruction shifts a register contents a specified distance (*shift_magnitude*) and direction (based on syntax and/or *shift_magnitude* sign) while preserving the sign bit of the original number. This instruction provides arithmetic shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on **The Science Dictionary** site are helpful.

The versions of this instruction using `ashift` syntax support the following shift operations:

- For a positive *shift_magnitude*, `ashift` produces an arithmetic shift right with sign bit preservation. The sign bit value back-fills the left-most bit positions vacated by the arithmetic shift right.
- For a negative *shift_magnitude*, `ashift` produces a logical shift left, but does not guarantee sign bit preservation. If the negative *shift_magnitude* is too large, the `ashift` operation saturates the destination register. A logical shift left that would otherwise lose non-sign bits off the left-hand side saturates to the maximum positive or negative value instead.

NOTE: One may view the `ashift` operation as a multiplication or division operation. Viewed this way, the *shift_magnitude* is the power of 2 multiplied by the *src_reg* number. Positive magnitudes cause multiplication ($N \times 2^n$), and negative magnitudes produce division ($N \times 2^{-n}$ or $N / 2^n$).

The versions of this instruction using >>> syntax only support arithmetic right shift operations using positive *shift_magnitude* values.

The versions of this instruction using <<< syntax only support logical left shift operations using positive *shift_magnitude* values.

The **Arithmetic Shift (Accumulator Destination Register) Operations** table provide more detailed information about arithmetic shift operations.

Table 8-18: Arithmetic Shift (Accumulator Destination Register) Operations

Table 8-19:

Syntax	Description
">>>", <<<, and ashift	The value in *src_reg* is shifted by the number of places specified in *shift_magnitude*, and the result is stored into *dest_reg*. The ashift versions can shift 40-bit accumulator registers by up to –32 through +31 places.

The *dest_reg* and *src_reg* may be a 40-bit register.

For 40-bit *src_reg*, valid shift magnitudes are –32 through +31, zero included.

The accumulator versions shift all 40 bits of those registers.

The accumulator versions of this instruction always implicitly modify the *src_reg* values.

This is a **32-bit instruction** and can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	**AV0S**	**AV0**
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AShiftAcc Example

```
/* AShiftAcc syntax summary */
/* a0 = ashift a0 by Dreg_lo ; arithmetic/logical shift */
/* a1 = ashift a1 by Dreg_lo ; arithmetic/logical shift */
/* a0 = a0 <<< 0 ; logical shift left */
/* a0 = a0 <<< UImm5 ; logical shift left */
```

```
/* a0 = a0 >>> UImm5N ; arithmetic shift right */
/* a1 = a1 <<< 0 ; logical shift left */
/* a1 = a1 <<< UImm5 ; logical shift left */
/* a1 = a1 >>> UImm5N ; arithmetic shift right */
/ * AShiftAcc syntax examples */
a0 = a0 >>> 1 ; /* arithmetic right shift, accumulator */
a0 = ashift a0 by r7.l ; /* shift, accumulator */
a1 = ashift a1 by r7.l ; /* shift, accumulator */
```

16-Bit Logical Shift (LShift16)

General Form

Dsp32Shf
DREG_L = lshift DREG_L by DREG_L
DREG_L = lshift DREG_H by DREG_L
DREG_H = lshift DREG_L by DREG_L
DREG_H = lshift DREG_H by DREG_L
Dsp32ShfImm
DREG_L = DREG_L LHSH4
DREG_L = DREG_H LHSH4
DREG_H = DREG_L LHSH4
DREG_H = DREG_H LHSH4

Abstract

This instruction shifts a register half by the specified number of bits and returns the shifted value.

See Also (LShift16Vec)

LShift16 Description

The logical shift low/high half data register destination instruction shifts a register contents a specified distance (*shift_magnitude*) and direction (based on syntax and/or *shift_magnitude* sign), discarding any bits shifted out of the register and backfilling vacated bits with zeros. This instruction provides logical shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on **The Science Dictionary** site are helpful.

The versions of this instruction using `lshift` syntax support the following shift operations:

- For a positive *shift_magnitude*, `lshift` produces an logical shift right, discarding any bits shifted out of the register and backfilling vacated bits with zeros.
- For a negative *shift_magnitude*, `lshift` produces a logical shift left, discarding any bits shifted out of the register and backfilling vacated bits with zeros.

NOTE: Shift magnitudes that exceed the size of the destination register produce all zeros in the result. For example, shifting a 16-bit register value by 20 bit places (a valid operation) produces 0x0000.

The versions of this instruction using >> syntax only support logical shift right operations using positive *shift_magnitude* values.

The versions of this instruction using << syntax only support logical shift left operations using positive *shift_magnitude* values.

The **Logical Shift (16 Bit Destination Register) Operations** table provide more detailed information about logical shift operations.

Table 8-20: Logical Shift (16 Bit Destination Register) Operations

Table 8-21:

Syntax	Description
>>, <<, and lshift	The value in *src_reg* is shifted by the number of places specified in *shift_magnitude*, and the result is stored into *dest_reg*. The lshift versions can shift 16-bit half data registers by up to –16 through +15 places.

The *dest_reg* and *src_reg* may be a 16-bit half data register.

For 16-bit *src_reg*, valid shift magnitudes are –16 through +15, zero included.

The data register versions of this instruction shift 16 bits for half-word registers.

The half data register versions of this instruction do not implicitly modify the *src_reg* values. Optionally, *dest_reg* may be the same data register as *src_reg*. Doing this explicitly modifies the source register.

This **32-bit instructions** can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

LShift16 Example

```
/* LShift16 syntax summary */
/* DDST_Lo_Hi = lshift DSRC1_Lo_Hi by DSRC0_Lo ; logical shift */
/* DDST_Lo_Hi = DSRC_Lo_Hi << 0 ; logical shift left */
/* DDST_Lo_Hi = DSRC_Lo_Hi << UImm4 ; logical shift lef */
/* DDST_Lo_Hi = DSRC_Lo_Hi >> UImm4N ; logical shift right */
/* LShift16 syntax examples */
r3.l = r0.l >> 4 ; /* logical shift right, half-word register */
```

```
r3.l = r0.h >> 4 ; /* logical shift right; half-word register combinations are
arbitrary */
r3.h = r0.l << 12 ; /* logical shift left, half-word register */
r3.h = r0.h << 14 ; /* logical shift left; half-word register combinations are
arbitrary */
r3.l = lshift r0.l by r2.l ; /* logical shift, direction controlled by sign of R2.L */
r3.h = lshift r0.l by r2.l ;
/* If r0.h = -64 (or 0xFFC0), then performing . . . */
r3.h = r0.h >> 4 ;
/* . . . produces r3.h = 0x0FFC (or 4092), losing the sign */
```

Vectored 16-Bit Logical Shift (LShift16Vec)

General Form

Dsp32Shf
DREG = lshift DREG by DREG_L (v)
Dsp32ShfImm
DREG = DREG LHSH4 (v)

Abstract

This instruction shifts a 16-bit vector left or right by the value in the XOP register.

See Also (LShift16)

LShift16Vec Description

The logical shift data register destination (vector) instruction performs two independent shifts, shifting a contents of a register's low half and high half a specified distance (*shift_magnitude*) and direction (based on syntax and/or *shift_magnitude* sign), discarding any bits shifted out of the two half registers and backfilling vacated bits with zeros. This instruction provides logical shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on **The Science Dictionary** site are helpful.

The versions of this instruction using lshift syntax support the following shift operations:

- For a positive *shift_magnitude*, lshift produces an logical shift right, discarding any bits shifted out of the register and backfilling vacated bits with zeros.
- For a negative *shift_magnitude*, lshift produces a logical shift left, discarding any bits shifted out of the register and backfilling vacated bits with zeros.

NOTE: Shift magnitudes that exceed the size of the destination register produce all zeros in the result. For example, shifting a 16-bit register value by 20 bit places (a valid operation) produces 0x0000.

The versions of this instruction using >> syntax only support logical shift right operations using positive *shift_magnitude* values.

The versions of this instruction using << syntax only support logical shift left operations using positive *shift_magnitude* values.

The **Logical Shift (16 Bit Destination Register) Operations** table provide more detailed information about logical shift operations.

Table 8-22: Logical Shift (16 Bit Destination Register) Operations

Table 8-23:

Syntax	Description
>>, <<, and lshift	The value in *src_reg* is shifted by the number of places specified in *shift_magnitude*, and the result is stored into *dest_reg*. The lshift versions can shift 16-bit half data reg isters by up to –16 through +15 places.

The *dest_reg* and *src_reg* may be a 32-bit half data register. The shift operations are applied to the 16-bit half registers within the *src_reg*.

For 16-bit *src_reg*, valid shift magnitudes are –16 through +15, zero included.

The data register versions of this instruction shift 16 bits for half-word registers.

The half data register versions of this instruction do not implicitly modify the *src_reg* values. Optionally, *dest_reg* may be the same data register as *src_reg*. Doing this explicitly modifies the source register.

This **32-bit instructions** can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

LShift16Vec Example

```
/* LShift16Vec syntax summary */
/* DDST = lshift DSRC1 by DSRC0_L (v) ; logical shift, vector (dual) */
/* DDST = DSRC << 0 (v) ; logical shift left, vector (dual) */
/* DDST = DSRC << UImm4 (v) ; logical shift left, vector (dual) */
/* DDST = DSRC >> UImm4N (v) ; logical shift right, vector (dual) */
/* LShiftVec syntax examples */
r4=r5>>3 (v) ; /* logical right shift immediate R5.H and R5.L by 3 bits */
r4=r5<<3 (v) ; /* logical left shift immediate R5.H and R5.L by 3 bits */
r2=lshift r7 by r5.l (v) ;
/* logically shift (right or left, depending on sign of r5.l) R7.H and R7.L by
magnitude of R5.L */
```

32-Bit Logical Shift (LShift)

General Form

ALU2op
DREG >>= DREG
DREG <<= DREG
Logi2Op
DREG >>= uimm5
DREG <<= uimm5
Dsp32Shf
DREG = lshift DREG by DREG_L
Dsp32ShfImm
DREG = DREG LSH5

Abstract

This instruction shifts a register by the specified number of bits and returns the shifted value.

LShift Description

The logical shift data register destination instruction shifts a register contents a specified distance (*shift_magnitude*) and direction (based on syntax and/or *shift_magnitude* sign), discarding any bits shifted out of the register and backfilling vacated bits with zeros. This instruction provides logical shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on **The Science Dictionary** site are helpful.

The versions of this instruction using `lshift` syntax support the following shift operations:

- For a positive *shift_magnitude*, `lshift` produces an logical shift right, discarding any bits shifted out of the register and backfilling vacated bits with zeros.
- For a negative *shift_magnitude*, `lshift` produces a logical shift left, discarding any bits shifted out of the register and backfilling vacated bits with zeros.

NOTE: Shift magnitudes that exceed the size of the destination register produce all zeros in the result. For example, shifting a 16-bit register value by 20 bit places (a valid operation) produces 0x0000.

The versions of this instruction using >>= and >> syntax only support logical shift right operations using positive *shift_magnitude* values.

The versions of this instruction using <<= and << syntax only support logical shift left operations using positive *shift_magnitude* values.

The **Logical Shift (16 Bit Destination Register) Operations** table provide more detailed information about logical shift operations.

Table 8-24: Logical Shift (16 Bit Destination Register) Operations

Table 8-25:

Syntax	Description
>>= and <<=	The value in *dest_reg* is shifted by the number of places specified by *shift_magnitude*. The data size is always 32 bits long. The entire 32 bits of the *shift_magnitude* determine the shift value. Shift magnitudes larger than 0x1F produce a 0x00000000 result.
>>, <<, and lshift	The value in *src_reg* is shifted by the number of places specified in *shift_magnitude*, and the result is stored into *dest_reg*. The lshift versions can shift 32-bit data reg isters by up to –32 through +31 places.

The *dest_reg* and *src_reg* may be a 32-bit data register.

For 32-bit *src_reg*, valid shift magnitudes are –32 through +31, zero included.

The data register versions of this instruction shift 32 bits for word registers.

The data register versions of this instruction do not implicitly modify the *src_reg* values. Optionally, *dest_reg* may be the same data register as *src_reg*. Doing this explicitly modifies the source register.

The versions of this instruction using >>, <<, and lshift syntax are **32-bit instructions**, which can sometimes save execution time (over a 16-bit encoded instruction) because they can be issued in parallel with certain other instructions.

The versions of this instruction using >>= and <<= syntax are **16-bit instructions** (which takes up less memory space over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

LShift Example

```
/* LShift syntax summary */
/* DDST >>= DSRC ; logical shift right */
/* DDST <<= DSRC ; logical shift left */
/* DDST >>= SRCI ; logical shift right */
/* DDST <<= SRCI ; logical shift left */
/* DDST = lshift DSRC1 by DSRC0_L ; logical shift */
/* DDST = DSRC << 0 ; logical shift left */
/* DDST = DSRC << UImm5 ; logical shift left */
/* DDST = DSRC >> UImm5N ; logical shift right */
/* LShift syntax examples */
r3 >>= 17 ; /* logical shift right */
r3 <<= 17 ; /* logical shift left */
r3 = r6 >> 4 ; /* logical shift right, 32-bit word */
r3 = r6 << 4 ; /* logical shift left, 32-bit word */
r3 >>= r0 ; /* logical shift right */
r3 <<= r1 ; /* logical shift left */
```

Accumulator Logical Shift (LShiftA)

General Form

Dsp32Shf
a0 = lshift a0 by DREG_L
a1 = lshift a1 by DREG_L
Dsp32ShfImm
a0 = a0 LSH5
a1 = a1 LSH5

Abstract

This instruction shifts an accumulator left by the specified number of bits and returns the shifted value.

LShiftA Description

The logical shift accumulator register destination instruction shifts a register contents a specified distance (*shift_magnitude*) and direction (based on syntax and/or *shift_magnitude* sign), discarding any bits shifted out of the register and backfilling vacated bits with zeros. This instruction provides logical shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on **The Science Dictionary** site are helpful.

The versions of this instruction using `lshift` syntax support the following shift operations:

- For a positive *shift_magnitude*, `lshift` produces an logical shift right, discarding any bits shifted out of the register and backfilling vacated bits with zeros.
- For a negative *shift_magnitude*, `lshift` produces a logical shift left, discarding any bits shifted out of the register and backfilling vacated bits with zeros.

NOTE: Shift magnitudes that exceed the size of the destination register produce all zeros in the result. For example, shifting a 16-bit register value by 20 bit places (a valid operation) produces 0x0000.

The versions of this instruction using >> syntax only support logical shift right operations using positive *shift_magnitude* values.

The versions of this instruction using << syntax only support logical shift left operations using positive *shift_magnitude* values.

The **Logical Shift (Accumulator Destination Register) Operations** table provide more detailed information about logical shift operations.

Table 8-26: Logical Shift (Accumulator Destination Register) Operations

Table 8-27:

Syntax	Description
>>, <<, and lshift	The value in *src_reg* is shifted by the number of places specified in *shift_magnitude*, and the result is stored into *dest_reg*. The lshift versions can shift 40-bit accumulator reg isters by up to –32 through +31 places.

The *dest_reg* and *src_reg* must be the same 40-bit accumulator register.

For 32-bit *src_reg*, valid shift magnitudes are –32 through +31, zero included.

The accumulator versions shift all 40 bits of those registers.

The accumulator versions of this instruction always implicitly modify the *src_reg* values.

This **32-bit instructions** can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	**AV0**
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

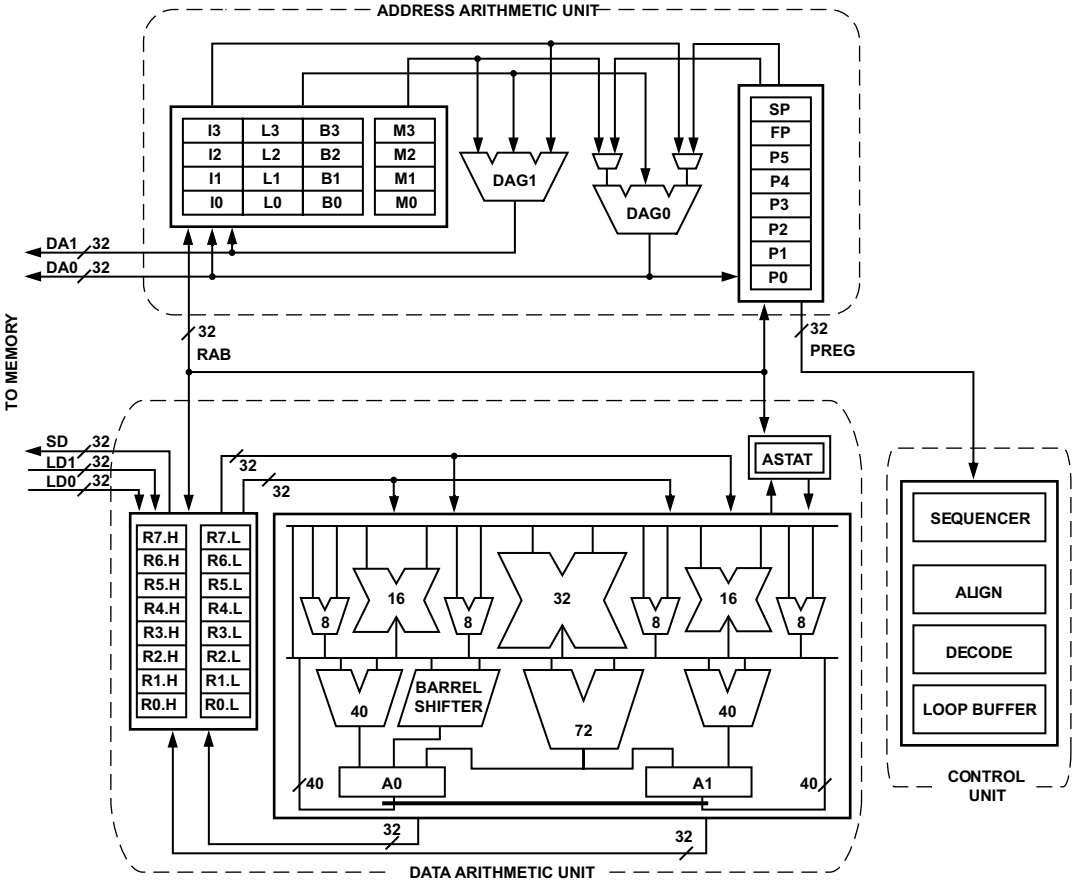
LShiftA Example

```
/* LShiftA syntax summary */
/* a0_1 = lshift a0_1 by DSRC0_L ; logical shift */
/* a0_1 = a0_1 << 0 ; logical shift left */
/* a0_1 = a0_1 << UImm5 ; logical shift left */
/* a0_1 = a0_1 >> UImm5N ; logical shift right */
/* LShiftA syntax examples */
a0 = a0 >> 7 ; /* Accumulator right shift */
a1 = a1 >> 25 ; /* Accumulator right shift */
a0 = a0 << 7 ; /* Accumulator left shift */
a1 = a1 << 14 ; /* Accumulator left shift */
```

```
a0 = lshift a0 by r7.l ;
a1 = lshift a1 by r7.l ;
```

Sequencer Instructions

The sequencer instructions provide program flow control operations, which execute on the **control unit** in the processor core. Users can take advantage of these instructions to force new values into the program counter and change program flow, branch conditionally, set up loops, and call and return from subroutines.



The operation types of sequencer instructions include:

- Branch Operations
- Control Code Bit Management Operations
- Event Management Operations
- Stack Operations
- Synchronization Operations

Branch Operations

These operations provide branching of program flow operations, unconditionally or with conditional operands:

- Conditional Jump Immediate (BrCC)
- Jump (Jump)
- Jump Immediate (JumpAbs)
- Call (Call)
- Return from Branch (Return)
- Hardware Loop Set Up (LoopSetup)

Conditional Jump Immediate (BrCC)

General Form

BrCC
if !cc jump imm10s2
if !cc jump imm10s2 (bp)
if cc jump imm10s2
if cc jump imm10s2 (bp)

Abstract

The Jump instruction forces a new value into the Program Counter (PC) to change program flow. This branches based on the value of the CC status bit. The (bp) option helps the processor improve branch instruction performance. The default is branch predicted-not-taken.

See Also (Jump, JumpAbs)

BrCC Description

The branch CC (conditional jump) instruction forces a new value into the Program Counter (PC) to change the program flow, based on the value of the CC bit.

- For if CC, a CC bit = 1 causes a branch to an address, computed by adding the signed, even offset to the current PC value.
- For if !cc, a cc bit = 0 causes a branch to an address, computed by adding the signed, even relative offset to the current PC value.

The range of valid offset values for the jump is –1024 through 1022.

The branch prediction option, (bp), helps the processor improve branch instruction performance. The default is branch predicted-not-taken. By appending (bp) to the instruction, the branch becomes predicted-taken. When dynamic branch prediction is enabled the (bp) option is used as an initial value for the dynamic predictor when the branch is first learned.

Typically, code analysis shows that a good default condition is to predict branch-taken for branches to a prior address (backwards branches), and to predict branch-not-taken for branches to subsequent addresses (forward branches).

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

BrCC Example

```
if cc jump 0xFFFFFE08 (bp) ;
/* offset is negative in 11 bits, so target address is a backwards branch, branch
predicted */
if cc jump 0x0B4 ;
/* offset is positive, so target offset address is a forwards branch, branch not
predicted */
if !cc jump 0xFFFFFC22 (bp) ;
/* negative offset in 11 bits, so target address is a backwards branch, branch
predicted */
if !cc jump 0x120 ;
/* positive offset, so target address is a forwards branch, branch not predicted */
if cc jump dest_label ;
/* assembler resolved target, abstract offsets */
```

Jump (Jump)

General Form

ProgCtrl
jump (PREG)
jump (pc+ PREG)

Abstract

The Jump instruction forces a new value into the Program Counter (PC) to change program flow.

See Also (BrCC, JumpAbs)

Jump Description

The jump pointer instruction forces a new value into the Program Counter (PC) to change program flow.

The new address may be *indirect* (provided by a pointer register) or may be *indexed* (PC plus an offset provided by a pointer register). In the indirect and indexed versions of the instruction, the value in the pointer register (Preg) must be an even number (bit 0 of the register =0) to maintain 16-bit address alignment. Otherwise, an odd offset in the pointer register causes the processor to generate a misaligned instruction fetch exception.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

Jump Example

```
jump (p5) ;
/* P5 contains the absolute address of the target */
jump (pc + p2) ;
/* P2 relative absolute address of the target and then a presentation of the absolute
values for target */
```

Jump Immediate (JumpAbs)

General Form

UJump
jump.s imm12nxs2
CallA
jump.l imm24s2
Jump32
jump.a buimm32
jump.xl bimm32

Abstract

The Jump instruction forces a new value into the Program Counter (PC) to change program flow.

See Also (BrCC, Jump)

JumpAbs Description

The jump absolute instruction forces a new value into the Program Counter (PC) to change program flow.

The assembler accepts the syntax `jump`*`label`* which translated to a 16-bit, 32-bit or 64-bit instruction with PC-relative offset as required.

The instruction size may be specified explicitly by a postfix: `jump.s` is a 16-bit instruction with 12-bit signed PC-relative offset. `jump.l` is a 32-bit instruction with a 12-bit signed PC-relative offset. `jump.xl` is a 64-bit instruction with an 32-bit PC-relative offset. `jump.x` is an alternative form of the variable length instruction.

The `jump.a` form of the instruction sets the PC to the 32-bit absolute value encoded in the instruction.

This instruction encodes as a **16-bit instruction**, **32-bit instruction**, or **64-bit instruction**, depending on the size of the offset value. This instruction may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

JumpAbs Example

```
jump get_new_sample ;
/* assembler resolved target, abstract offsets */
jump 0x224 ;
/* offset is positive in 13 bits, so target address is PC + 0x224, a forward jump */
jump.s 0x224 ;
```

```
/* same as above with jump “short” syntax */
jump.l 0xFFFACE86 ;
/* offset is negative in 25 bits, so target address is PC + 0x1FA CE86, a backwards
jump */
```

Call (Call)

General Form

ProgCtrl
call (PREG)
call (pc+ PREG)
CallA
call.l imm24nxs2
Jump32
call.a buimm32
call.xl bimm32

Abstract

The Call instruction branches to the address specified and then updates the RETS register with the address of the instruction directly following the Call instruction.

See Also (Return)

Call Description

The CALL instruction calls a subroutine from an address that may be *indirect* (provided by a pointer register), may be *indexed* (PC plus an offset provided by a pointer register), may be a *label* (a program label that provides a signed, even, PC-relative offset), or may be an *immediate value* (provides a signed, even, PC-relative offset). In the indirect and indexed versions of the instruction, the value in the pointer register (Preg) must be an even number (bit 0 of the register =0) to maintain 16-bit address alignment. Otherwise, an odd offset in the pointer register causes the processor to generate a misaligned instruction fetch.

The assembler accepts the syntax `call label` which is translated to a 16-bit, 32-bit or 64-bit instruction with PC-relative offset as required. The instruction size of the PC-relative form may be specified explicitly by a postfix: `call.l` is a 32-bit instruction with a 12-bit signed PC-relative offset. `call.xl` is a 64-bit instruction with an 32-bit PC-relative offset. `call.x` is an alternative form of the variable length instruction.

The `call.a` form of the instruction sets the PC to the 32-bit absolute value encoded in the instruction.

After the CALL instruction executes and execution of the subroutine is completed, the program sequencer resumes program execution at the instruction address pointed to by the RETS register. The address write to the RETS register occurs when the CALL instruction is committed. Even when used as the last instruction of a loop, the CALL instruction functions correctly. If the CALL were placed at a loop end, the RETS register contains the loop top address.

This instruction encodes as a **16-bit instruction**, **32-bit instruction**, or **64-bit instruction**, depending on the size of the offset value. This instruction may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

Call Example

```
call ( p5 ) ;
call ( pc + p2 ) ;
call 0x123456 ;
call get_next_sample ;
```

Return from Branch (Return)

General Form

ProgCtrl
rts
rti
rtx
rtn
rte

Abstract

Each of these instructions branch to the address specified in their return registers. The interrupt return instructions will also clear their interrupts corresponding bit in the IPEND register.

See Also (Call)

Return Description

The return instruction forces a return from a subroutine, maskable interrupt or NMI routine, exception routine, or emulation routine. The **Types of Return Instructions** table provides a description of the operations provided by each type of return. Note that the interrupt return instructions also clear their interrupt's corresponding bit in the IPEND register.

Table 8-28: Types of Return Instructions

Table 8-29:

Mnemonic	Description
RTS	Forces a return from a subroutine by loading the value of the RETS register into the Program Counter (PC), causing the processor to fetch the next instruction from the address contained in RETS. For nested subroutines, you must save the value of the RETS Register. Otherwise, the next subroutine CALL instruction overwrites it.
RTI	Forces a return from an interrupt routine by loading the value of the RETI register into the PC. When an interrupt is generated, the processor enters a non-interruptible state. Saving RETI to the stack re-enables interrupt detection so that subsequent, higher priority interrupts can be serviced (or nested) during the current interrupt service routine. If RETI is not saved to the stack, higher priority interrupts are recognized but not serviced until the current interrupt service routine concludes. Restoring RETI back off the stack at the conclusion of the interrupt service routine masks subsequent interrupts until the RTI instruction executes. In any case, RETI is protected against inadvertent corruption by higher priority interrupts.
RTX	Forces a return from an exception routine by loading the value of the RETX register into the PC.
RTN	Forces a return from a non-maskable interrupt (NMI) routine by load- ing the value of the RETN register into the PC.

Table 8-29:

Mnemonic	Description
RTE	Forces a return from an emulation routine and emulation mode by load- ing the value of the RETE register into the PC. Because only one emulation routine can run at a time, nesting is not an issue, and saving the value of the RETE register is unnecessary.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

The **Required Modes for Return Instructions** table identifies the modes required for each return instruction.

Table 8-30: Required Modes for Return Instructions

Mnemonic	Required Mode
RTS	User and Supervisor
RTI, RTX, and RTN	Supervisor only. Any attempt to execute in User mode produces a protection violation exception.
RTE	Emulation only. Any attempt to execute in User mode or Supervi- sor mode produces an exception.

Return Example

```
rts ;
rti ;
rtx ;
rtn ;
rte ;
```

Hardware Loop Set Up (LoopSetup)

General Form

LoopSetup
lsetup (uimm4s2o4 , uimm10s2o4) LC [1]
lsetup (uimm4s2o4 , uimm10s2o4) LC = PREG [2]
lsetup (uimm4s2o4 , uimm10s2o4) LC = PREG >>1 [3]
LoopSetup
lsetup (uimm10s2o4) LC = uimm10
lsetupz (uimm10s2o4) LC = PREG
lsetupz (uimm10s2o4) LC = PREG >> 1
lsetuplez (uimm10s2o4) LC = PREG
lsetuplez (uimm10s2o4) LC = PREG >> 1

1. Provides encoding for: `LOOP loop_name LC0 ; LOOP_BEGIN loop_name ; LOOP_END loop_name ;`
2. Provides encoding for: `LOOP loop_name LC0 = Preg ; LOOP_BEGIN loop_name ; LOOP_END loop_name ;`
3. Provides encoding for: `LOOP loop_name LC0 = Preg >> 1 ; LOOP_BEGIN loop_name ; LOOP_END loop_name ;`

Abstract

The zero-overhead loop set up instruction provides a flexible, count-based, hardware loop mechanism, implementing efficient, zero-overhead software loops. The term "zero-overhead" means the software does not incur a performance or code size penalty by decrementing the loop counter, evaluating a loop condition, calculating the target address, and branching to the address.

See Also (none)

LoopSetup Description

The zero-overhead loop setup instruction provides a flexible, counter- based, hardware loop mechanism that provides efficient, zero-overhead software loops. In this context, zero-overhead means that the software in the loops does not incur a performance or code size penalty by decrementing a counter, evaluating a loop condition, then calculating and branching to a new target address.

NOTE: When the *Begin_Loop* address is the next sequential address after the `LSETUP` instruction, the loop has zero overhead. If the *Begin_Loop* address is not the next sequential address after the `LSETUP` instruction, there is some overhead that is incurred on loop entry only.

The architecture includes two sets of three registers each to support two independent, nestable loops. The registers are *Loop_Top* (LTx), *Loop_Bottom* (LBx) and *Loop_Count* (LCx). The LT0, LB0, and LC0 registers describe *Loop0*, and the LT1, LB1, and LC1 registers describe *Loop1*.

The LOOP and LSETUP instructions permit initializing all three registers using a single instruction. The size of the LOOP and LSETUP instructions only supports a finite number of bits, so the loop range is limited. However, LT0 and LT1, LB0 and LB1 and LC0 and LC1 can be initialized manually using move instructions if loop length and repetition count need to be beyond the limits supported by the LOOP and LSETUP syntax. A single loop (initialized using this method) can span the entire 4G bytes of memory space.

NOTE: When initializing LT0 and LT1, LB0 and LB1, and LC0 and LC1 manually, make sure that *Loop_Top* (LTx) and *Loop_Bottom* (LBx) are configured before setting *Loop_Count* (LCx) to the desired loop count value.

The instruction syntax supports an optional initialization value from a pointer register (Preg) or pointer register divided by 2.

NOTE: The LOOP, LOOP_BEGIN, LOOP_END legacy syntax from previous Blackfin processors is supported by the Blackfin+ processor assembler.

The legacy syntax is encoded as LSETUP syntax, which contains the same information in a more compact form.

If LCx is nonzero when the fetch address equals LBx, the processor decrements LCx and places the address in LTx into the PC. The loop always executes once through because *Loop_Count* is evaluated at the end of the loop.

There are two special cases for small loop count values. A value of 0 in Loop_Count causes the hardware loop mechanism to neither decrement or loopback, causing the instructions enclosed by the loop pointers to be executed as straight-line code. A value of 1 in *Loop_Count* causes the hardware loop mechanism to decrement only (not loopback), also causing the instructions enclosed by the loop pointers to be executed as straight-line code.

In the instruction syntax, the designation of the loop counter–LC0 or LC1– determines which loop level is initialized. Consequently, to initialize *Loop0*, code LC0; to initialize *Loop1*, code LC1.

In the case of nested loops that end on the same instruction, the processor requires *Loop0* to describe the outer loop and *Loop1* to describe the inner loop. The user is responsible for meeting this requirement.

For example, if LB0=LB1, then the processor assumes loop 1 is the inner loop and loop 0 the outer loop.

Just like entries in any other register, loop register entries can be saved and restored. If nesting beyond two loop levels is required, the user can explicitly save the outermost loop register values, re-use the registers for an inner loop, and then restore the outermost loop values before terminating the inner loop. In such a case, remember that loop 0 must always be outside of loop 1. Alternately, the user can implement the outermost loop in software with the Conditional Jump structure.

Begin_Loop, the value loaded into LTx, is a 5-bit, PC-relative, even offset from the current instruction to the first instruction in the loop. The user is required to preserve half-word alignment by maintaining even

values in this register. The offset is interpreted as a one's-complement, unsigned number, eliminating backwards loops.

End_Loop, the value loaded into LBx, is an 11-bit, unsigned, even, PC-relative offset from the current instruction to the last instruction of the loop. When using the LSETUP instruction, *Begin_Loop* and *End_Loop* are typically address labels. The linker replaces the labels with offset values.

A loop counter register (LC0 or LC1) counts the trips through the loop. The register contains a 32-bit unsigned value, supporting as many as 4,294,967,294 trips through the loop. The loop is disabled (subsequent executions of the loop code pass through without reiterating) when the loop counter equals 0.

If no *LoopStartLabel* is specified then the loop start is implied to be the instruction following the LSETUP instruction.

The Z suffix (LSETUPZ), means that the entire loop will be skipped if the count starts at zero. The LEZ suffix (LSETUPLEZ) means that entire loop will be skipped if the starting count is Less than or Equal to Zero. When a LSETUPZ instruction with a loop count of zero commits the LSBit of it's associated LT register will be set. This is used to mark this as an lsetupz should we interrupt the loop. The end of the loop will clear the bit.

It is important to understand the following LSETUP operations and how these affect loop operations:

- If a start address is specified in the LSETUP instruction, the address is a 5-bit, PC-relative, unsigned, even offset (4 to 30) address. If a start address is not specified in the LSETUP instruction, the address used is the address of the instruction following the LSETUP instruction. The absolute start address is computed and stored in LT0 or LT1 register on LSETUP instruction commit.
- The end address is an 11-bit, PC-relative, unsigned, even offset (4 through 2046) address. The absolute end address is computed and stored in the LB0 or LB1 register on LSETUP instruction commit.
- The values in the loop counter 0 (LC0) and loop counter 1 (LC1) registers are treated as 32-bit unsigned values, except in the LSETUPLEZ version of the instruction. For LSETUPLEZ, the loop count value is treated as a signed value. The value in the LC0 or LC1 register decrements each time a loop bottom instruction is executed, until the count reaches 0. When executing a loop end instruction, when the value in LC0 or LC1 is not 0 or 1, a loop back operation occurs.
- A loop is disabled when the its loop count (LC0 or LC1) equals 0.
- The sequencer treats a **constant** loop count of zero as special and loads the counter with the value 0xffffffff.

LoopSetup Example

```
/* examples for three-part loop setup ... */
/* LOOP loop_name loop_counter */
/* LOOP_BEGIN loop_name */
/* LOOP_END loop_name */

loop MyRepeatedOperations LC0 ; /* define loop 'MyRepeatedOperations' with Loop
Counter 0 */
```

```
loop_begin MyRepeatedOperations ; /* place before the first instruction in the loop */
   nop;
loop_end MyRepeatedOperations ; /* place after the last instruction in the loop */

loop MyOtherRepeatedOperations LC1 ; /* define loop 'MyOtherRepeatedOperations' with
Loop Counter 1 */
loop_begin MyOtherRepeatedOperations ; /* place before the first instruction in the
loop */
   nop;
loop_end MyOtherRepeatedOperations ; /* place after the last instruction in the loop */

/* a loop with a specified beginning offset */
loop loop_2 lc0 = p0;
loop_begin loop_2;
   nop;
loop_end loop_2;

/* a loop without a specified beginning offset */
loop lc0 = 45;
   nop;
loop_end;

/* note that loopz and looplez opcodes to lsetupz and lsetuplez */

/* examples for single line loop setup ... */
/* LSETUP (Begin_Loop, End_Loop) Loop_Counter */

lsetup ( 4, 4 ) lc0 ;
lsetup ( poll_bit, end_poll_bit ) lc0 ;
lsetup ( 4, 6 ) lc1 ;
lsetup ( FIR_filter, bottom_of_FIR_filter ) lc1 ;
lsetup ( 4, 8 ) lc0 = p1 ;
lsetup ( 4, 8 ) lc0 = p1>>1 ;
```

Control Code Bit Management Operations

These operations provide control code bit operations needed to support condtional operations:

- Move CC to a D Register (CCToDreg)
- Move CC To/From ASTAT (CCToStat16)
- Move to CC (MvToCC)
- Move Status to CC (MvToCC_STAT)
- 32-Bit Register Compare and Set CC (CompRegisters)
- Accumulator Compare and Set CC (CompAccumulators)
- 32-Bit Pointer Register Compare and Set CC (CCFlagP)

Move CC to a D Register (CCToDreg)

General Form

`CC2Dreg`
`DREG` = cc
`DREG` = !cc

Abstract

This instruction moves CC to a 32-bit D Register. The register will either be 1 on 0.

CCToDreg Description

The move CC to data register instruction moves *either* the condition code (CC) bit value *or* moves the negated CC bit value to a data register.

When copying the CC bit value into a 32-bit register, the operation moves the CC bit value into the least significant bit of the register, zero-extended to 32 bits. The two cases for `Dreg = CC` are as follows.

- If CC = 0, the data register becomes 0x00000000.
- If CC = 1, the data register becomes 0x00000001.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

CCToDreg Example

```
r0 = cc ;
r1 =! cc ;
```

Move CC To/From ASTAT (CCToStat16)

General Form

CC2Stat
CB = cc
CB |= cc
CB &= cc
CB ^= cc

Abstract

This instruction moves CC to another ASTAT bit. It is illegal to use the CC bit as source and destination in the same instruction (for example, CC=CC or CC&=CC.

See Also (MvToCC, MvToCC_STAT)

CCToStat16 Description

The move CC to arithmetic status register instruction sets or clears status bits based on the logic operations and the status of the condition code (CC) bit.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	**AC1**	**AC0**	...	...	...	RND_MOD	...	**AQ**	**CC**	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CCToStat16 Example

```
az = cc ; /* status bit equals cc */
an |= cc ; /* status bit equals status bit OR cc */
```

```
ac0 &= cc ; /* status bit equals status bit AND cc */
av0 ^= cc ; /* status bit equals status bit XOR cc */
```

Move to CC (MvToCC)

General Form

CC2Dreg
cc = DREG
cc = !cc

Abstract

This instruction either moves an OR of all bits in the data register to the CC bit or negates the CC bit.

See Also (CCToStat16, MvToCC_STAT)

MvToCC Description

The move data register to CC instruction *either* moves an OR of all bits in the data register *or* moves the negated state of the condition code (CC) bit to the CC bit. When copying a data register to the CC bit, the operation sets the CC bit to 1 if any bit in the source data register is set; that is, if the register is nonzero. Otherwise, the operation clears the CC bit.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	**CC**	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvToCC Example

```
cc = r4 ;
cc = !cc ;
```

Move Status to CC (MvToCC_STAT)

General Form

CC2Stat
cc = CB
cc |= CB
cc &= CB
cc ^= CB

Abstract

This instruction moves a status bit or LSB of a register to CC. It is illegal to use the CC bit as source and destination in the same instruction (for example, CC=CC or CC&=CC are illegal).

See Also (CCToStat16, MvToCC)

MvToCC_STAT Description

The move status bit to CC instruction sets or clears the condition code (CC) bit based on the logic operations and the status of the status bits.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvToCC_STAT Example

```
cc = av1 ;  /* cc equals status bit */
cc |= aq ;  /* cc equals cc OR status bit */
```

```
cc &= an ;  /* cc equals cc AND status bit */
cc ^= ac1 ; /* cc equals cc XOR status bit */
```

32-Bit Pointer Register Compare and Set CC (CCFlagP)

General Form

CCFlag
cc = PREG == PREG
cc = PREG == imm3
cc = PREG < PREG
cc = PREG < imm3
cc = PREG <= PREG
cc = PREG <= imm3
cc = PREG < PREG (iu)
cc = PREG < uimm3 (iu)
cc = PREG <= PREG (iu)
cc = PREG <= uimm3 (iu)

Abstract

This instruction compares two pointer registers.

See Also (CompRegisters, CompAccumulators)

CCFlagP Description

The compare pointer and set CC instruction sets or clears the condition code (CC) bit based on a comparison of two values. The input operands are pointer registers (Preg).

The compare operations are nondestructive on the input operands and affect only the CC bit. The value of the CC bit determines all subsequent conditional branching.

The various forms of the compare pointer instruction perform 32-bit signed compare operations on the input operands or an unsigned compare operation (if the (IU) optional mode is appended). The compare operations perform a subtraction and discard the result of the subtraction without affecting user registers. The compare operation that you specify determines the value of the CC bit.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CCFlagP Example

```
cc = p3 == p2 ; /* equal, register, signed */
cc = p0 == 1 ; /* equal, immediate, signed */
cc = p0 < p3 ; /* less than, register, signed */
cc = p2 < -4 ; /* less than, immediate, signed */
cc = p1 <= p0 ; /* less than or equal, register, signed */
cc = p4 <= 3 ; /* less than or equal, immediate, signed */
cc = p5 < p3 (iu) ; /* less than, register, unsigned */
cc = p1 < 0x7 (iu) ; /* less than, immediate, unsigned */
cc = p2 <= p0 (iu) ; /* less than or equal, register, unsigned */
cc = p3 <= 2 (iu) ; /* less than or equal, immediate unsigned */
```

Accumulator Compare and Set CC (CompAccumulators)

General Form

CCFlag
cc = a0 == a1
cc = a0 < a1
cc = a0 <= a1

Abstract

This instruction compares the two accumulators and sets CC.

See Also (CompRegisters, CCFlagP)

CompAccumulators Description

The compare accumulator instruction sets the condition code (CC) bit based on a comparison of two values. The input operands are Accumulators.

These instructions perform 40-bit signed compare operations on the Accumulators. The compare operations perform the subtraction A0–A1 and discard the result of the subtraction without affecting user registers. The compare operation that you specify determines the value of the CC bit.

No unsigned compare operations or immediate compare operations are performed for the Accumulators. The compare operations are nondestructive on the input operands, and affect only the CC bit and the status bits. All subsequent conditional branching is based on the value of the CC bit.

The Compare Accumulator instruction uses the values shown in the **Compare Accumulator Instruction Values** table in compare operations after the A0–A1 subtraction is performed.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

Table 8-31: Compare Accumulator Instruction Values

Table 8-32:

Comparison	Signed
Equal	AZ =1
Less than	AN =1

Table 8-32:

Comparison	Signed
Less than or equal	AN or AZ =1

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_ MOD	...	AQ	**CC**	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CompAccumulators Example

```
cc = a0 == a1 ; /* equal, signed */
cc = a0 < a1 ;  /* less than, accumulator, signed */
cc = a0 <= a1 ; /* less than or equal, accumulator, signed */
```

32-Bit Register Compare and Set CC (CompRegisters)

General Form

CCFlag
cc = DREG == DREG
cc = DREG == imm3
cc = DREG < DREG
cc = DREG < imm3
cc = DREG <= DREG
cc = DREG <= imm3
cc = DREG < DREG (iu)
cc = DREG < uimm3 (iu)
cc = DREG <= DREG (iu)
cc = DREG <= uimm3 (iu)

Abstract

This instruction compares two 32-bit registers and sets CC or sets CC if a register is non-zero.

See Also (CompAccumulators, CCFlagP)

CompRegisters Description

The compare data register instruction sets the condition code (CC) bit based on a comparison of two values. The input operands are D-registers.

The compare operations are nondestructive on the input operands and affect only the CC bit and the status bits. The value of the CC bit determines all subsequent conditional branching.

The various forms of the Compare Data Register instruction perform 32-bit signed compare operations on the input operands or an unsigned compare operation, if the (IU) optional mode is appended. The compare operations perform a subtraction and discard the result of the subtraction without affecting user registers. The compare operation that you specify determines the value of the CC bit.

The Compare Data Register instruction uses the values shown in the **Compare Data Register Values** table in signed and unsigned compare operations.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

Table 8-33: Compare Data Register Values

Table 8-34:

Comparison	Signed	Unsigned
Equal	AZ=1	n/a
Less than	AN=1	AC0=0
Less than or equal	AN or AZ=1	AC0=0 or AZ=1

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	**AC0**	...	...	...	RND_ MOD	...	AQ	**CC**	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CompRegisters Example

```
cc = r3 == r2 ; /* equal, register, signed */
cc = r7 == 1 ; /* equal, immediate, signed */
/* If r0 = 0x8FFF FFFF and r3 = 0x0000 0001, then the signed operation . . . */
cc = r0 < r3 ; /* less than, register, signed */
/* . . . produces cc = 1, because r0 is treated as a negative value */
cc = r2 < -4 ; /* less than, immediate, signed */
cc = r6 <= r1 ; /* less than or equal, register, signed */
cc = r4 <= 3 ; /* less than or equal, immediate, signed */
/* If r0 = 0x8FFF FFFF and r3 = 0x0000 0001,then the unsigned operation . . . */
cc = r0 < r3 (iu) ; /* less than, register, unsigned */
/* . . . produces CC = 0, because r0 is treated as a large unsigned value */
cc = r1 < 0x7 (iu) ; /* less than, immediate, unsigned */
cc = r2 <= r0 (iu) ; /* less than or equal, register, unsigned (a) */
cc = r3 <= 2 (iu) ; /* less than or equal, immediate unsigned (a) */
```

Event Management Operations

These operations provide interrupt and exception related operations:

- Interrupt Control (IMaskMv)
- Emulator Exception (EmuExcpt)
- Raise Interrupt (Raise)

Interrupt Control (IMaskMv)

General Form

ProgCtrl
cli DREG
sti DREG

Abstract

The CLI instruction disables general interrupts, and the STI instruction enables interrupts.

IMaskMv Description

The enable interrupts instruction (`sti`) globally enables interrupts by restoring the previous state of the interrupt system from a data register into the `IMASK` register.

The disable interrupts instruction (`cli`) globally disables general interrupts by clearing the `IMASK` register to all zeros. In addition, the instruction copies the previous contents of `IMASK` into a user-specified register in order to save the state of the interrupt system. The disable interrupts instruction does not mask NMI, reset, exceptions, and emulation.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

The enable interrupts and disable interrupts instructions executes only in **Supervisor mode**. If execution is attempted in User mode, the instruction produces an Illegal Use of Protected Resource exception.

IMaskMv Example

```
sti r3 ; /* previous state of IMASK restored from Dreg */
cli r3 ; /* previous state of IMASK moved to Dreg (a) */
```

Emulator Exception (EmuExcpt)

General Form

ProgCtrl
emuexcpt

Abstract

The EMUEXCPT instruction allows processor to enter emulation mode.

Mode Description

The force emulation instruction forces an emulation exception, allowing the processor to enter emulation mode. When emulation is enabled, the processor immediately takes an exception into emulation mode. When emulation is disabled, `EMUEXCPT` behaves the same as a `NOP` instruction. The emulation exception is the highest priority event in processor.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**

Mode Example

```
emuexcpt ;
```

Raise Interrupt (Raise)

General Form

ProgCtrl
raise uimm4
excpt uimm4

Abstract

The EXCPT instruction forces the specified exception (range 0 through 15).

Raise Description

The force interrupt / reset / exception instruction forces a specified interrupt or reset or exception to occur. Typically, it is a software method of invoking a hardware event for debug purposes.

When the RAISE instruction is issued, the processor sets a bit in the ILAT register corresponding to the interrupt vector specified by the *uimm4* constant in the instruction. The interrupt executes when its priority is high enough to be recognized by the processor. The RAISE instruction causes these events to occur given the uimm4 arguments shown in the **uimm4 Arguments and Events** table.

When the EXCPT instruction is issued, the sequencer vectors to the exception handler that the user provides. Application-level code uses the force exception instruction for operating system calls. The instruction does not set the EVSW bit (bit 3) of the ILAT register.

Table 8-35: uimm4 Arguments and Events

Table 8-36:

uimm4	Event
0	reserved
1	RST
2	NMI
3	reserved
4	reserved
5	IVHW
6	IVTMR
7	IVG7
8	IVG8

Table 8-36:

uimm4	Event
9	IVG9
10	IVG10
11	IVG11
12	IVG12
13	IVG13
14	IVG14
15	IVG15

The `RAISE` instruction cannot invoke exception (EXC) or emulation (EMU) events. Use the `EXCPT` and `EMUEXCPT` instructions, respectively, for those events.

The `RAISE` instruction does not take effect before the write-back stage in the pipeline.

This **16-bit instruction** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

The force interrupt / reset / exception instruction executes only in *Supervisor mode*. If execution is attempted in User mode, the force interrupt / reset instruction produces an Illegal Use of Protected Resource exception.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	**CC**	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Raise Example

```
raise 1 ; /* Invoke RST */
raise 6 ; /* Invoke IVTMR timer interrupt */
excpt 4 ;
```

Stack Operations

These operations provide memory stack management operations:

- Linkage (Linkage)
- Stack Pop (Pop)
- Stack Push (Push)
- Stack Push/Pop Multiple Registers (PushPopMul16)

Linkage (Linkage)

General Form

Linkage
link uimm16s4
unlink

Abstract

The linkage instruction controls the stack frame space on the stack and the Frame Pointer (FP) for that space. LINK allocates the space and UNLINK de-allocates the space.

Linkage Description

The linkage instruction controls the stack frame space on the stack and the frame pointer (FP) for that space. LINK allocates the space and UNLINK de-allocates the space.

LINK saves the current RETS and FP registers to the stack, loads the FP register with the new frame address, then decrements the stack pointer (SP) by the user-supplied frame size value.

Typical applications follow the LINK instruction with a push multiple instruction to save pointer and data registers to the stack.

The user-supplied argument for LINK determines the size of the allocated stack frame. LINK always saves RETS and FP on the stack, so the minimum frame size is 2 words when the argument is zero. The maximum stack frame size is 218 + 8 = 262152 bytes in 4-byte increments.

UNLINK performs the reciprocal of LINK, de-allocating the frame space by moving the current value of FP into SP and restoring previous values into FP and RETS from the stack.

The UNLINK instruction typically follows a pop multiple instruction that restores pointer and data registers previously saved to the stack.

The frame values remain on the stack until a subsequent push, push multiple or LINK operation overwrites them.

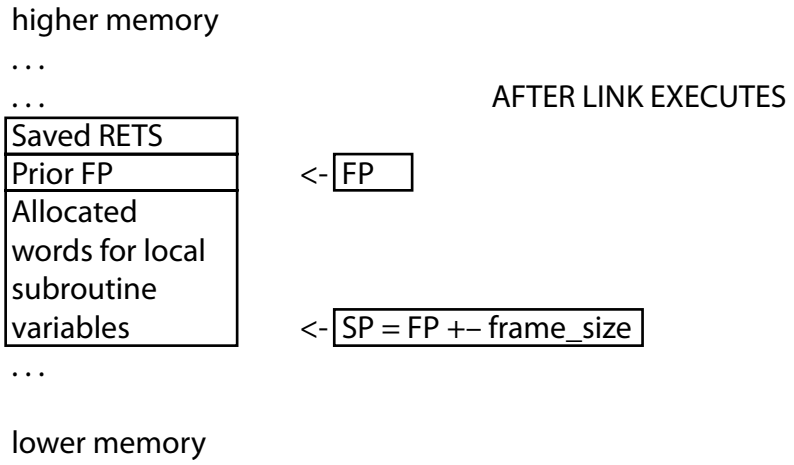
To preserve stack integrity, the FP*must not be modified* by user code between LINK and UNLINK execution.

Neither LINK nor UNLINK may be interrupted. Exceptions that occur while either of these instructions are executing cause the instruction to abort. For example, a load operation or a store operation might cause a protection violation while LINK is executing. In that case, SP and FP are reset to their original values prior to the execution of this instruction. This measure ensures that the instruction can be restarted after the exception.

Note that when a LINK operation aborts due to an exception, the stack memory may already be changed due to stores that have already completed before the exception. Similarly, an aborted UNLINK operation may leave the FP and RETS registers changed because of a load that has already completed before the interruption.

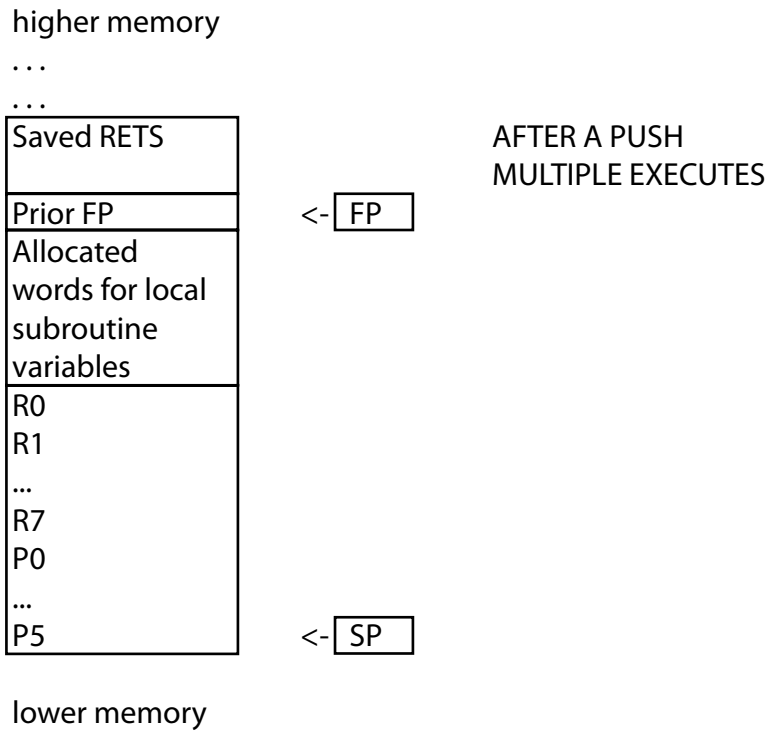
The series of illustrations show how the stack contents change. After executing a LINK instruction, the stack contains (for example) the contents shown in the **Stack After Link Executes** figure.

Figure 8-9: Stack After Link Executes



Following the `LINK` and a push multiple instruction, the stack contains (for example) the contents shown in the **Stack After Push Multiple Executes** figure.

Figure 8-10: Stack After Push Multiple Executes



The stack pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts, as described above.

This **32-bit instruction** may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

Linkage Example

```
link 8 ; /* establish frame with 8 words allocated for local variables */
[ -- sp ] = (r7:0, p5:0) ; /* save D- and P-registers */
(r7:0, p5:0) = [ sp ++ ] ; /* restore D- and P-registers */
unlink ; /* close the frame* /
```

Stack Pop (Pop)

General Form

PushPopReg
POPREG = [sp++]

Abstract

This instruction loads the contents of the stack indexed by the current stack pointer into a specified register.

See Also (Push, PushPopMul16)

Pop Description

The pop instruction loads the contents of the stack—indexed by the current stack pointer (SP)—into a specified register. The instruction post-increments the stack pointer to the next occupied location in the stack before concluding.

The stack grows down from high memory to low memory, therefore the decrement operation is used for pushing, and the increment operation is used for popping values. The stack pointer always points to the last used location. When a pop operation is issued, the value pointed to by the stack pointer is transferred and the SP is replaced by SP + 4.

The following series of illustrations show what the stack would look like when a pop such as `R3 = [ SP ++ ]` occurs.

Figure 8-11: Stack Beginning State

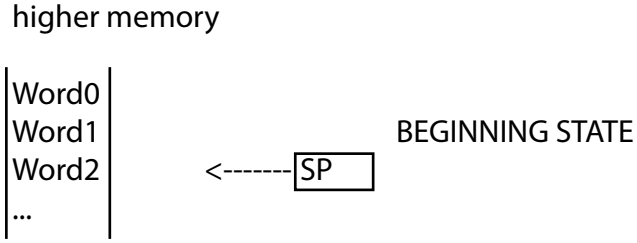


Figure 8-12: Load Register From Stack

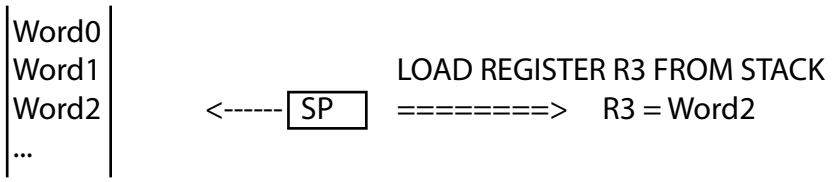
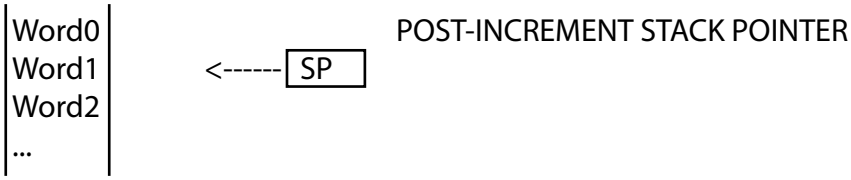


Figure 8-13: Post-Increment Stack Pointer



This **16-bit instruction** may be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode** for most cases, but explicit access to `USP`, `SEQSTAT`, `SYSCFG`, `RETI`, `RETX`, `RETN`, `RETE`, and `EMUDAT` requires Supervisor mode. A protection violation exception results if any of these registers are explicitly accessed from User mode.

The `ASTAT = [SP++]` version of this instruction explicitly affects arithmetic status bits. Status bits are not affected by other versions of this instruction.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	**AV1S**	**AV1**	**AV0S**	**AV0**
...	...	**AC1**	**AC0**	...	...	...	RND_ MOD	...	**AQ**	**CC**	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Pop Example

```
r0 = [sp++] ; /* Load Data Register instruction */
p4 = [sp++] ; /* Load Pointer Register instruction */
i1 = [sp++] ; /* Pop instruction */
reti = [sp++] ; /* Pop instruction; supervisor mode required */
```

Stack Push (Push)

General Form

PushPopReg
[--sp] = PUSHREG

Abstract

This instruction stores the contents of a specified register in the stack.

See Also (Pop, PushPopMul16)

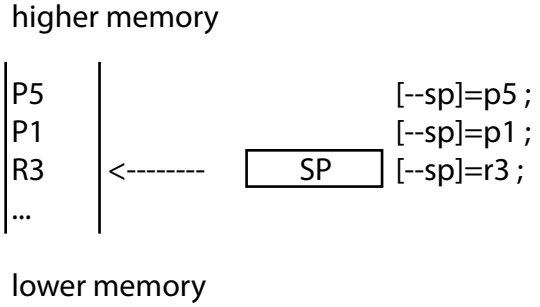
Push Description

The push instruction stores the contents of a specified register in the stack. The instruction pre-decrements the stack pointer (SP) to the next available location in the stack first. Push and push multiple are the only instructions that perform pre-modify functions.

The stack grows down from high memory to low memory. Consequently, the decrement operation is used for pushing, and the increment operation is used for popping values. The stack pointer always points to the last used location. Therefore, the effective address of the push is SP - 4.

The following illustration shows what the stack would look like when a series of pushes occur.

Figure 8-14: Stack Following a Series of Pushes



The stack pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts.

Push/pop on RETS has no effect on the interrupt system.

Push/pop on RETI does affect the interrupt system.

Pushing RETI enables the interrupt system, whereas popping RETI disables the interrupt system.

Pushing the stack pointer is meaningless since it cannot be retrieved from the stack. Using the stack pointer as the destination of a pop instruction (as in the fictional instruction `SP = [SP++]`) causes an undefined instruction exception.

This **16-bit instruction** may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode** for most cases, but explicit access to `USP`, `SEQSTAT`, `SYSCFG`, `RETI`, `RETX`, `RETN`, `RETE`, and `EMUDAT` requires Supervisor mode. A protection violation exception results if any of these registers are explicitly accessed from User mode.

Push Example

```
[ -- sp ] = r0 ;
[ -- sp ] = r1 ;
[ -- sp ] = p0 ;
[ -- sp ] = i0 ;
```

Stack Push/Pop Multiple Registers (PushPop-Mul16)

General Form

PushPopMult
(PREG_RANGE) = [sp++]
(DREG_RANGE) = [sp++]
(DREG_RANGE , PREG_RANGE) = [sp++]
[--sp] = (PREG_RANGE)
[--sp] = (DREG_RANGE)
[--sp] = (DREG_RANGE , PREG_RANGE)

Abstract

This instruction pushes or pops the contents of multiple data and/or pointer registers to or from the stack.

See Also (Pop, Push)

PushPopMul16 Description

The push multiple instruction saves the contents of multiple data and/or pointer registers to the stack, and the pop multiple instruction restores the contents of multiple data and/or pointer registers from the stack. The range of registers to be pushed (saved) or popped (restored) always includes the highest index data register (R7) and/or highest index pointer register (P5) plus any contiguous lower index registers specified by the user down to and including R0 and/or P0.

Push Multiple Instruction Operations

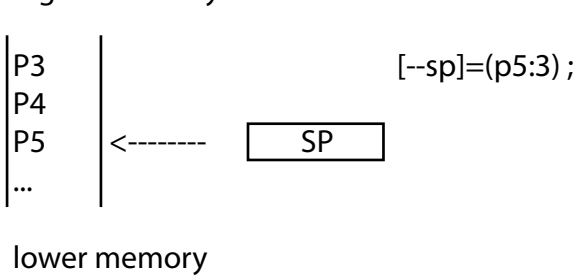
The push multiple instruction operations start by saving the register having the lowest index then advance to the register with the highest index. The index of the first register saved in the stack is specified by the user in the instruction syntax. Data registers are pushed before Pointer registers if both are specified in one instruction.

The push multiple instruction pre-decrements the stack pointer to the next available location in the stack first.

NOTE: Push and Push Multiple are the only instructions that perform pre-modify functions.
The stack grows down from high memory to low memory, therefore the decrement operation is the same used for pushing, and the increment operation is used for popping values. The stack pointer always points to the last used location, making the effective address of the push is SP - 4.

The **Stack Following a Push Multiple** illustration shows what the stack would look like when a push multiple occurs.

Figure 8-15: Stack Following a Push Multiple



Because the push multiple instruction always saves the lowest-indexed registers first, it is advisable that a runtime system be defined to have its compiler scratch registers as the lowest- indexed registers. For instance, data registers `R0`, `P0` would be the return value registers for a simple calling convention.

Pop Multiple Instruction Operations

The pop multiple instruction operations start by restoring the register having the highest index then descend to the register with the lowest index. The index of the last register restored from the stack is specified by the user in the instruction syntax. Pointer registers are popped before data registers, if both are specified in the same instruction.

The instruction post-increments the stack pointer to the next occupied location in the stack before concluding.

The stack grows down from high memory to low memory, therefore the decrement operation is used for pushing, and the increment operation is used for popping values. The Stack Pointer always points to the last used location. When a pop operation is issued, the value pointed to by the Stack Pointer is transferred and the `SP` is replaced by `SP + 4`.

The series of illustrations show how the stack appears when a pop multiple such as `(R7:5) = [SP++]` occurs.

Figure 8-16: Stack Beginning State of Pop Multiple

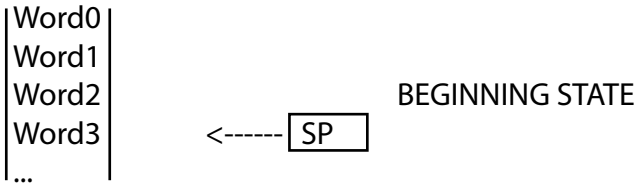


Figure 8-17: Load Register R7 From Stack during Pop Multiple

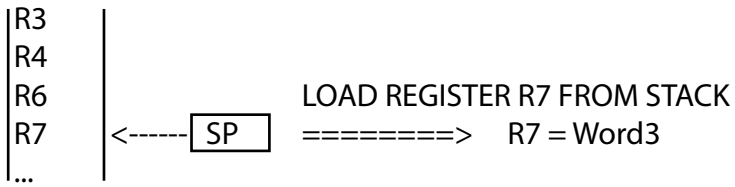


Figure 8-18: Load Register R6 From Stack during Pop Multiple

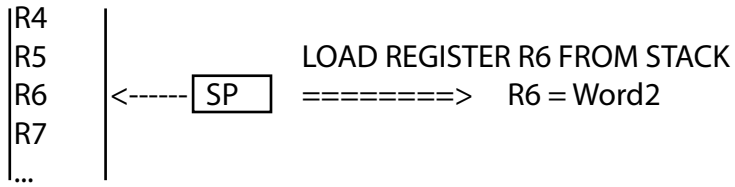


Figure 8-19: Load Register R5 From Stack during Pop Multiple

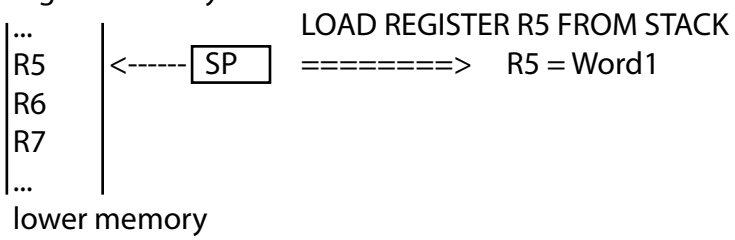
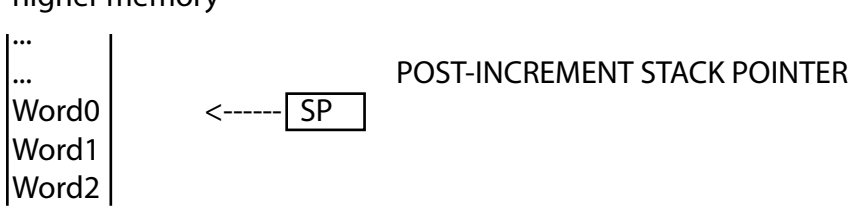


Figure 8-20: Post-Increment Stack Pointer after Pop Multiple



The value(s) just popped remain on the stack until another push instruction overwrites it.

The intended usage for the pop multiple instruction is to recover register values that were previously pushed onto the stack. The user must exercise programming discipline to restore the stack values back to their intended registers from the first-in, last-out structure of the stack. Pop exactly the same registers that were pushed onto the stack, but pop them in the opposite order.

Push Multiple and Pop Multiple Common Features

Although the PushPopMul16 instruction takes a variable amount of time to complete (depending on the number of registers to be saved/restored), the instruction reduces compiled code size.

This instruction is not interruptible. Interrupts asserted after the first issued stack write operation (for push) or stack read operation (for pop) are appended until all the writes or reads complete. However, exceptions that occur while this instruction is executing cause it to abort gracefully. For example:

- While push multiple is executing, a load/store operation might cause a protection violation. In that case, the SP is reset to its value before the execution of this instruction. Note that when a push multiple operation is aborted due to an exception, the memory state is changed by the stores that have already completed before the exception.
- While pop multiple is executing, a load/store operation might cause a protection violation In that case, the SP is reset to its original value prior to the execution of this instruction. Note that when a pop multiple operation aborts due to an exception, some of the destination registers are changed as a result of loads that have already completed before the exception.

These measures ensures that the instruction can be restarted after the exception.

The stack pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts.

Only data register (R7-0) and pointer registers (P5-0) may be operands for this instruction. The SP and FP registers may not be used as operands for this instruction.

This **16-bit instruction** may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

PushPopMul16 Example

```
/* push multiple examples */
[ -- sp ] = (r7:5, p5:0) ; /* D-registers R4:0 excluded */
[ -- sp ] = (r7:5, p5:0) ; /* D-registers R4:0 excluded */
[ -- sp ] = (r7:2) ; /* R1:0 excluded */
[ -- sp ] = (p5:4) ; /* P3:0 excluded */

/* pop multiple examples */
(p5:4) = [ sp ++ ] ; /* P3 through P0 excluded */
(r7:2) = [ sp ++ ] ; /* R1 through R0 excluded */
(r7:5, p5:0) = [ sp ++ ] ; /* D-registers R4 through R0 optionally excluded */
```

Synchronization Operations

These operations provide processor synchronization operations:

- Cache Control (CacheCtrl)
- Sync (Sync)
- SyncExcl (SyncExcl)
- NOP (NOP)
- 32-Bit No Operation (NOP32)
- TestSet (TestSet)

Cache Control (CacheCtrl)

General Form

CacheCtrl
prefetch [PREGA]
flushinv [PREGA]
flush [PREGA]
iflush [PREGA]

Abstract

These instructions provide the ability to manipulate the caches. The prefetch causes the data cache to prefetch the cache line associated with the effective address provided as the contents of the p-register.

See Also (Sync)

CacheCtrl Description

These instructions provide the ability to manipulate the cachesa;

- prefetch causes the data cache to prefetch the cache line associated with the effective address provided as the contents of the p-register.
- flushinv causes the data cache to invalidate a particular line in the cache.
- flush causes the a line of data in the cache to be syncronized with higher levels of memory.
- iflush causes the instruction cache to invalidate a particular line in the cache.

prefetch Instruction Operations

The Data Cache Prefetch instruction causes the data cache to prefetch the cache line that is associated with the effective address in the P-register. The operation causes the line to be fetched if it is not currently in the data cache and if the address is cacheable (that is, if bit CPLB_L1_CHBL = 1). If the line is already in the cache or if the cache is already fetching a line, the prefetch instruction performs no action, like a NOP.

This instruction may generate CPLB exceptions. For example, exception 0x26 can be generated upon execution of the PREFETCH[P0] instruction if P0 points to an invalid memory location. However, external memory will not be accessed when any of these exceptions are generated.

The instruction can post-increment the line pointer by the cache line size.

flushinv Instruction Operations

The Data Cache Line Invalidate instruction causes the data cache to invalidate a specific line in the cache. The contents of the P-register specify the line to invalidate. If the line is in the cache and dirty, the cache line

is written out to the next level of memory in the hierarchy. If the line is not in the cache, the instruction performs no action, like a NOP.

This instruction may generate CPLB exceptions. For example, exception 0x26 can be generated upon execution of the FLUSHINV[P0] instruction if P0 points to an invalid memory location. However, external memory will not be accessed when any of these exceptions are generated. The instruction can post-increment the line pointer by the cache line size.

flush Instruction Operations

The Data Cache Flush instruction causes the data cache to synchronize the specified cache line with higher levels of memory. This instruction selects the cache line corresponding to the effective address contained in the P-register. If the cached data line is dirty, the instruction writes the line out and marks the line clean in the data cache. If the specified data cache line is already clean or the cache does not contain the address in the P-register, this instruction performs no action, like a NOP.

This instruction may generate CPLB exceptions. For example, exception 0x26 can be generated upon execution of the FLUSH[P0] instruction if P0 points to an invalid memory location. However, external memory will not be accessed when any of these exceptions are generated.

The instruction can post-increment the line pointer by the cache line size.

iflush Instruction Operations

The Instruction Cache Flush instruction causes the instruction cache to invalidate a specific line in the cache. The contents of the P-register specify the line to invalidate. The instruction cache contains no dirty bit. Consequently, the contents of the instruction cache are never flushed to higher levels.

This instruction does not cause address exception violations. If a protection violation associated with the address occurs, the instruction acts as a NOP and does not cause a protection violation exception.

The instruction can post-increment the line pointer by the cache line size.

CacheCtrl Instruction Common Features

These instructions have post-modify versions, where the value in the pointer register (Preg) used as address for the prefetch and flush is incremented by the cache-line size.

These instructions do not cause address exception violations. If the effective address is misaligned or outside the allowed memory region, these instructions have no effect.

This **16-bit instruction** may not be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either **User or Supervisor mode**.

CacheCtrl Example

```
prefetch [ p2 ] ;
prefetch [ p0 ++ ] ;
flushinv [ p2 ] ;
flushinv [ p0 ++ ] ;
flush [ p2 ] ;
flush [ p0 ++ ] ;
```

```
iflush [ p2 ] ;
iflush [ p0 ++ ] ;
```

Sync (Sync)

General Form

ProgCtrl
idle
csync
ssync
sti idle DREG

Abstract

The instructions are DSYNC (Data Sync), SSYNC (System Sync), CSYNC (Core Sync), IDLE, and STI IDLE.

See Also (CacheCtrl)

Sync Description

The sync instructions (CSYNC, SSYNC, DSYNC, IDLE, and STI IDLE) provide the means to synchronize core, system, and data operations across all clock domains of the processor.

idle Instruction Operations

Typically, the IDLE instruction is part of a sequence to place the Blackfin processor in a quiescent state so that the external system can switch between core clock frequencies.

The first instruction following the IDLE is the first instruction to execute when the processor recovers from idle mode.

NOTE: Blackfin+ processors (unlike previous on previous Blackfin processors) an IDLE instruction is not required immediately following an SSYNC instruction.

csync Instruction Operations

The core synchronize (CSYNC) instruction ensures resolution of all pending core operations and the flushing of the core store buffer before proceeding to the next instruction. Pending core operations include any speculative states (for example, branch prediction) or exceptions. The core store buffer lies between the processor and the L1 cache memory.

CCYNC is typically used after core memory-mapped register writes to prevent imprecise behavior, unless otherwise specified in Blackfin Processor Programming Reference. For example, an SSYNC instruction is required to follow some core memory-mapped register accesses, such as when IMEM_CONTROL is written to while enabling cache.

Use CSYNC to enforce a strict execution sequence on loads and stores or to conclude all transitional core states before reconfiguring the core modes. For example, issue CSYNC before configuring memory-mapped registers (MMRs). CSYNC should also be issued after stores to memory-mapped registers to make sure the data reaches the memory-mapped register before the next instruction is fetched.

Typically, the Blackfin processor executes all load instructions strictly in the order that they are issued and all store instructions in the order that they are issued. However, for performance reasons, the architecture relaxes ordering between load and store operations. It usually allows load operations to access memory out of order with respect to store operations. Further, it usually allows loads to access memory speculatively. The core may later cancel or restart speculative loads. By using the core synchronize or system synchronize instructions and managing interrupts appropriately, you can restrict out-of-order and speculative behavior.

NOTE: Stores never access memory speculatively.

ssync Instruction Operations

The system synchronize (SSYNC) instruction forces all speculative, transient states in the core and system to complete before processing continues. Until SSYNC completes, no further instructions can be issued to the pipeline.

The SSYNC instruction performs the same function as core synchronize (CSYNC). In addition, SSYNC flushes any write buffers (between the L1 memory and the system interface) and generates a sync request signal to the external system. The operation requires an acknowledgement by the system before completing the instruction.

An SSYNC instruction should be used when ordering is required between a memory write and a memory read. For more information about these operations, see the memory or pointer instructions.

When strict ordering of instruction execution is required, by design, the Blackfin processor architecture allows reads to take priority over writes when there are no dependencies between the address that are accessed. In general, this execution order allows for increased performance. However, when an asynchronous memory device is mapped to a Blackfin processor, it is sometimes necessary to ensure the write occurs before the read. But, the Blackfin processor re-orders loads over stores if there is not a data dependency. In this case, an SSYNC between the write and read will ensure proper ordering is preserved.

NOTE: Blackfin+ processors (unlike previous on previous Blackfin processors) an IDLE instruction is not required immediately following an SSYNC instruction.

dsync Instruction Operations

The DSYNC instruction (data sync) is insures that all writes have completed to final destinations before any other writes commit.

sti idle Instruction Operations

The STI IDLE instruction (enable interrupts, then idle) performs an implicit SSYNC, then simultaneously restores IMASK from the specifed data register and enters idle mode.

Sync Instruction Common Features

These **16-bit instructions** may not be issued in parallel with certain other 16-bit instructions.

These instructions may be used in either **User or Supervisor mode**.

Sync Example

Example code sequence for IDLE

```
idle ;
```

Example code sequence for CSYNC---In this example, the CSYNC instruction ensures that the load instruction is not executed speculatively. CSYNC ensures that the conditional branch is resolved and any entries in the processor store buffer have been flushed. In addition, all speculative states or exceptions complete processing before CSYNC completes.

```
if cc jump away_from_here ;
/* produces speculative branch prediction */
csync ;
r0 = [p0] ; /* load */
```

Example code sequence for SSYNC---In this example, SSYNC ensures that the load instruction will not be executed speculatively. The instruction ensures that the conditional branch is resolved and any entries in the processor store buffer and write buffer have been flushed. In addition, all exceptions complete processing before SSYNC completes.

```
if cc jump away_from_here ;
/* produces speculative branch prediction */
ssync ;
r0 = [p0] ; /* load */
```

Example code sequence for DSYNC---In this example, DSYNC ensures that the load instruction will not be executed speculatively. The instruction ensures that the conditional branch is resolved and any entries in the processor store buffer and write buffer have been flushed. In addition, all exceptions complete processing before DSYNC completes.

```
if cc jump away_from_here ;
/* produces speculative branch prediction */
dsync ;
r0 = [p0] ; /* load */
```

Example code sequence for STI IDLE

```
sti idle r0 ;
```

SyncExcl (SyncExcl)

General Form

LdStExcl
syncexcl

Abstract

This instruction synchronizes the processor state with the exclusive state, capturing any pending write response and releasing exclusive memory access to a memory location.

SyncExcl Description

The SyncExcl (synchronize exclusive state) instruction forms the third part of the exclusive instruction sequence. It should be placed immediately after a store exclusive instruction to wait for the pending exclusive write response and set ASTAT.CC bit according to the response. After the syncexl has been executed, the ASTAT.CC bit contains 1 if the store exclusive successfully updated memory and contains 0 if the store exclusive did not successfully update. The syncexcl only updates th ASTAT.CC bit if there is actually a pending exclusive write transaction, so it may safely be used to save the response in code, saving the processor context.

The exclusive synchronize state instruction synchronize the processor state with the exclusive state, capturing any pending write response and releasing exclusive memory access to shareable memory space. For more information about illegal, non-shareable, and shareable memory regions, see Exclusive Memory Instructions.

The execution of an exclusive sychronize state instruction depends on the state of the processor and exclusive state at the start of the instruction. These status bits are listed in the **Exclusive Related Bits in Status Registers (SEQSTAT and ASTAT)** table. If the SEQSTAT.XMONITOR bit =1 and the SEQSTAT.XWACTIVE bit =1, the instruction stalls until the memory management unit (MMU) sets the SEQSTAT.XWAVAIL bit (=1). Then, (if SEQSTAT.XMONITOR, SEQSTAT.XWACTIVE, and SEQSTAT.XWAVAIL are set), the instruction copies the response from the exclusive write to the ASTAT.CC bit. On completion of the instruction, the SEQSTAT.XMONITOR, SEQSTAT.XWACTIVE, and SEQSTAT.XWAVAIL bits are all cleared, reseting the exclusive transaction state. If the instruction's execution is interrupted, the state is not updated.

Table 8-37: Exclusive Related Bits in Status Registers (SEQSTAT and ASTAT)

Table 8-38:

Flag	Condition	Value on completion of instruction	Meaning
ASTAT.CC	Only changed if XMONITOR and XWACTIVE at start	?	0=write failed, 1=write succeeded
SEQSTAT.XMONITOR	Always updated	0	
SEQSTAT.XWACTIVE	Always updated	0	
SEQSTAT.XWAVAIL	Always updated	0	

This **16-bit instruction** may not be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	**CC**	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

SyncExcl Example

```
syncexcl;  /* synchronize exclusive state */
```

NOP (NOP)

General Form

NOP16
nop

Abstract

This instruction increments the PC (and does nothing else).

See Also (NOP32)

NOP Description

The No Op instruction increments the PC and does nothing else.

Typically, the No Op instruction allows previous instructions time to complete before continuing with subsequent instructions. Other uses are to produce specific delays in timing loops or to act as hardware event timers and rate generators when no timers and rate generators are available.

This **16-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

NOP Example

```
nop ;
```

32-Bit No Operation (NOP32)

General Form

NOP32
mnop

Abstract

This instruction increments the PC (and does nothing else).

See Also (NOP)

NOP32 Description

The No Op instruction increments the PC and does nothing else.

Typically, the No Op instruction allows previous instructions time to complete before continuing with subsequent instructions. Other uses are to produce specific delays in timing loops or to act as hardware event timers and rate generators when no timers and rate generators are available.

MNOP can be used to issue loads or store instructions in parallel without invoking a 32-bit MAC or ALU operation.

This **32-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

NOP32 Example

```
mnop ;
mnop || /* a 16-bit instr. */ || /* a 16-bit instr. */ ;
```

TestSet (TestSet)

General Form

ProgCtrl
testset (PREGP)

Abstract

This instruction loads a byte, tests whether it is zero, then sets the most significant bit of the byte in memory. CC is set if the byte is originally zero, and cleared if the byte is originally nonzero. The sequence of memory transactions are atomic.

TestSet Description

The `testset` instruction is an atomic operation. (This sequence may be aborted by an interrupt, but will restart from the beginning upon return from interrupt. A byte protected in this manner may be used as a semaphore.) This instruction is primarily provided for backward compatability and it is recomended to use exclusive load and store instructions which make more efficient use of system resources (if that is possible). The `testset` instruction reads an indirectly addressed memory byte, tests whether it is zero, and then writes the byte back to memory with the most significant bit (MSB) set, all as one indivisible operation. If the byte is originally zero, the instruction sets the `CC` bit. If the byte is originally nonzero, the instruction clears the `CC` bit.

The `TESTSET` instruction is never executed speculatively. It is supported by bus-locked memory transactions on the system bus, so no other user of the bus, such as another core, can access memory between the test and set portions of this instruction. The `TESTSET` instruction can be interrupted by the core. If this happens the system bus is released and the `TESTSET` instruction is executed again upon return from the interrupt.

`TESTSET` should not be used in L1 SRAM or cacheable memory as its behavior in these regions varies between different derivatives. `TESTSET` must not be used with MMRs, I/O Device space or extended data access addresses.

This **16-bit instruction** may not be issued in parallel with certain other 16-bit instructions.

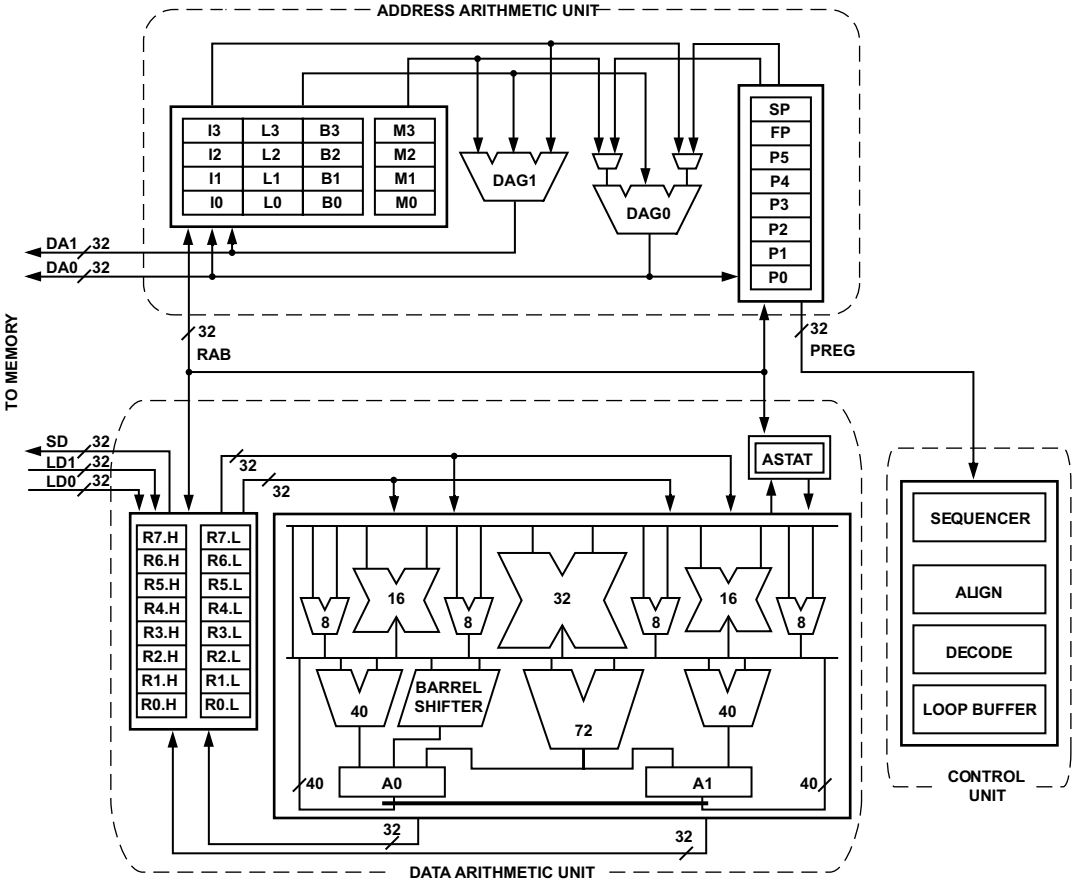
This instruction may be used in either **User or Supervisor mode**.

TestSet Example

```
testset (P3) ; /* test and set byte addressed by P3 */
```

Memory or Pointer Instructions

The memory and pointer instructions provide operations, which execute on the **address arithmetic unit** in the processor core. Users can take advantage of these instructions to load registers with data from memory locations, to store register data to memory locations, and to execute arithmetic operation using pointer registers.



The operation types of memory or pointer instructions include:

- Memory Load Operations
- Memory Store Operations
- Pointer Math Operations

Load from Immediate (Value) Operations

These operations provide register load operations on register and immediate value operands:

- 32-Bit Accumulator Register (.x) Initialization (LdImmToAxX)
- 32-Bit Accumulator Register (.w) Initialization (LdImmToAxW)
- Accumulator Register Initialization (LdImmToAx)
- 16-Bit Register Initialization (LdImmToDregHL)
- 32-Bit Register Initialization (LdImmToReg)
- Dual Accumulator 0 and 1 Registers Initialization (LdImmToAxDual)

Accumulator Register Initialization (LdImmToAx)

General Form

Dsp32Alu
a0 = 0
a1 = 0

Abstract

This instruction loads the accumulator register with the immediate value 0 (initializes the result register).

See Also (LdImmToAxX, LdImmToAxW)

LdImmToAx Description

The load immediate to accumulator instruction loads an immediate value (0) into an accumulator register. This operation initializes (or clears) the accumulator register.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdImmToAx Example

```
a0 = 0 ;
a1 = 0 ;
```

32-Bit Accumulator Register (.w) Initialization (LdImmToAxW)

General Form

LdImm
a0.w = imm32
a1.w = imm32

Abstract

This instruction initializes the lower 32-bits (.w) of the accumulator register from a 32-bit immediate value.

See Also (LdImmToAxX, LdImmToAx)

LdImmToAxW Description

The load immediate to accumulator 32-bit section instruction loads a immediate value, or explicit constant, into the the A0.w or A1.w register.

The instruction loads the 32-bit accumulator section from a 32-bit quantity, depending on the size of the immediate data.

The load operation uses the 32 bits of the input immediate value and leaves the unspecified portion of the accumulator register intact.

This **64-bit instruction** may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdImmToAxW Example

```
a0.w = 0x7FFFFFFF ;
a1.w = 0x80000000 ;
a0.w = MyResult ;
a1.w = MyOtherResult ;
```

32-Bit Accumulator Register (.x) Initialization (LdImmToAxX)

General Form

LdImm
a0.x = imm32
a1.x = imm32

Abstract

This instruction initializes the upper 8-bit (.x)of the accumulator register from a 32-bit immediate value.

See Also (LdImmToAxW, LdImmToAx)

LdImmToAxX Description

The load immediate to accumulator 8-bit section instruction loads a immediate value, or explicit constant, into the the A0.x or A1.x register.

The instruction loads the 8-bit accumulator section from a 32-bit quantity, depending on the size of the immediate data.

The load operation uses the least significant 8 bits of the input immediate value and leaves the unspecified portion of the accumulator register intact.

This **64-bit instruction** may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdImmToAxX Example

```
a0.x = 0x7FFFFFFF ;
a1.x = 0x80000000 ;
a0.x = MyResult ;
a1.x = MyOtherResult ;
```

16-Bit Register Initialization (LdImmToDregHL)

General Form

LdImmHalf
DST_L = imm16
DST_H = imm16

Abstract

This instruction loads a low-half register or a high-half register with a 16-bit immediate value.

LdImmToDregHL Description

The load immediate to high/low half register instruction loads an immediate value, or explicit constant, into a high or low half register.

The instruction loads a 16-bit quantity, depending on the size of the immediate data.

The 16-bit half-words are be loaded into either the high half or low half of a register. The load operation leaves the unspecified half of the register intact.

Loading a 32-bit value into a register using this load immediate instruction requires two separate instructions—one for the high and one for the low half. For example, to load the address `foo` into register `P3`, write:

```
p3.h = foo ;
p3.l = foo ;
```

The assembler automatically selects the correct half-word portion of the 32-bit literal for inclusion in the instruction word.

This **32-bit instruction** may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdImmToDregHL Example

```
r7.h = 63 ;
p3.l = 12 ;
i0.l = 4 ;
m2.h = 8 ;
l3.h = 0xbcde ;
```

32-Bit Register Initialization (LdImmToReg)

General Form

CompI2opD
DREG = imm7 (x)
CompI2opP
PREG = imm7 (x)
LdImmHalf
DST = imm16 (x)
DST = rimm16 (z)
LdImm
DREG = imm32
PREG = imm32
IREG = imm32
MREG = imm32
BREG = imm32
LREG = imm32
astat = imm32
rets = imm32
SYSREG2 = imm32
SYSREG3 = imm32

Abstract

This instruction initializes a 32-bit register to an immediate value. For the smaller instructions, where the immediate is less than 32, you can specify if you want the immediate value sign or zero extended to fill the register.

LdImmToReg Description

The load immediate to register instruction loads an immediate value, or explicit constants, into a register.

The instruction loads a 7-, 16-, or 32-bit quantity, depending on the size of the immediate data.

The zero-extended (z) versions of this instruction fill the upper bits of the destination register with zeros. The sign-extended (x) versions of this instruction fill the upper bits with the sign of the constant value.

The instruction opcode size varies with the immediate value size as follows:

- Load immediate to register of 7-bit data encodes as a **16-bit instruction**.
- Load immediate to register of 16-bit data encodes as a **32-bit instruction**.
- Load immediate to register of 7-bit data encodes as a **64-bit instruction**.

These load immediate instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdImmToReg Example

```
r7 = 63 (z) ;
p3 = 12 (z) ;
r0 = -344 (x) ;
r7 = 436 (z) ;
m2 = 0x89ab (z) ;
p1 = 0x1234 (z) ;
m3 = 0x3456 (x) ;
```

Dual Accumulator 0 and 1 Registers Initialization (LdImmToAxDual)

General Form

Dsp32Alu
a1 = a0 = 0

Abstract

This instruction loads the accumulator 0 and 1 registers (A0, A1) with the immediate value 0 (initializes both result registers).

LdImmToAxDual Description

The dual load immediate to accumulator instruction loads an immediate value (0) into both accumulator registers. This operation initializes (or clears) both of the accumulator registers.

This **32-bit instruction** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdImmToAxDual Example

```
a1 = a0 = 0 ;
```

Memory Load Operations

These operations provide memory load operations on register and immediate value operands:

- 8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)
- 16-Bit Load from Memory (LdM16bitToDregH)
- 16-Bit Load from Memory (LdM16bitToDregL)
- 32-Bit Load from Memory (LdM32bitToDreg)
- 16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
- 32-Bit Register Initialization (LdImmToReg)

8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)

General Form

LdSt
DREG = b[PREG ++] (z)
DREG = b[PREG --] (z)
DREG = b[PREG] (z)
DREG = b[PREG ++] (x)
DREG = b[PREG --] (x)
DREG = b[PREG] (x)
LdStIdxI
DREG = b[PREG + imm16reloc] (z)
DREG = b[PREG + imm16reloc] (x)
LdStAbs
DREG = b[uimm32] (z)
DREG = b[uimm32] (x)

Abstract

This instruction loads a register with an 8-bit value from memory. The value is sign or zero extended in the register.

See Also (LdM32bitToDreg, LdM16bitToDregH, LdM16bitToDregL, LdM16bitToDreg)

LdM08bitToDreg Description

The load byte to data register instruction loads an 8-bit byte value from a memory location into a 32-bit data register. The address of the memory location is identified with a pointer register, a pointer plus an offset, or a 32-bit absolute address. The byte value is sign-extended `(x)` or zero-extended `(z)` to 32 bits in the destination data register. The address used in this instruction has no restrictions for memory address alignment. This instruction supports the following options.

- Post-increment the source pointer by 1 byte `[Preg ++]`
- Post-decrement the source pointer by 1 byte `[Preg --]`
- Offset the source pointer with a 16-bit signed constant `[Preg + Offset]`

The instruction opcode size varies with the address type as follows:

- Load byte to register using a pointer register for the address encodes as a **16-bit instruction**.
- Load byte to register using a pointer register with 16-bit offset for the address encodes as a **32-bit instruction**.
- Load byte to register using a 32-bit absolute address encodes as a **64-bit instruction**.

The 16-bit load byte instructions may be issued in parallel with certain other instructions. The 32- and 64-bit load byte instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdM08bitToDreg Example

```
r3 = b [ p0 ] (z) ;
r7 = b [ p1 ++ ] (z) ;
r2 = b [ sp -- ] (z) ;
r0 = b [ p4 + 0xFFFF800F ] (z) ;
r3 = b [ p0 ] (x) ;
r7 = b [ p1 ++ ](x) ;
r2 = b [ sp -- ] (x) ;
r0 = b [ p4 + 0xFFFF800F ](x) ;
```

16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)

General Form

LdStPmod
DREG = w[PREG ++ PREG] (z)
DREG = w[PREG ++ PREG] (x)
LdSt
DREG = w[PREG ++] (z)
DREG = w[PREG --] (z)
DREG = w[PREG] (z)
DREG = w[PREG ++] (x)
DREG = w[PREG --] (x)
DREG = w[PREG] (x)
LdStII
DREG = w[PREG + uimm4s2] (z)
DREG = w[PREG + uimm4s2] (x)
LdStIdxI
DREG = w[PREG + imm16s2] (z)
DREG = w[PREG + imm16s2] (x)
LdStAbs
DREG = w[uimm32] (z)
DREG = w[uimm32] (x)

Abstract

This instruction loads a register with a 16-bit value from memory. The value is sign or zero extended in the register.

See Also (LdM08bitToDreg, LdM32bitToDreg, LdM16bitToDregH, LdM16bitToDregL)

LdM16bitToDreg Description

The load word to data register instruction loads a 16-bit value from a memory location into a 32-bit data register. The address of the memory location is identified with a pointer register, a pointer plus an offset, or a 32-bit absolute address. The word value is sign-extended (x) or zero-extended (z) to 32 bits in the destination data register. The address used in this instruction is restricted to even memory address align-

ment (2-byte half-word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source pointer by 2 bytes `[Preg ++]`
- Post-decrement the source pointer by 2 bytes `[Preg --]`
- Offset the source pointer with a 5-bit signed constant `[Preg + SmallOffset]`
- Offset the source pointer with a 16-bit signed constant `[Preg + LargeOffset]`
- Offset the source pointer with second pointer `[Preg ++ Preg]`

The syntax of the form:

```
Dest = w [ Src_1 ++ Src_2 ] ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dest = w [Src_1] ; /* load the 32-bit destination, indirect*/
Src_1 += Src_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- *Dest* is the destination register. (*Dreg* in the syntax example).
- *Src_1* is the first source register on the right-hand side of the equation.
- *Src_2* is the second source register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
r0 = w [p2 ++ p2] (z) ;
```

functions as:

```
r0 = w [p2] (z) ;
```

The instruction opcode size varies with the address type as follows:

- Load word to register using a pointer register for the address or using a pointer register with small offset for the address encodes as a **16-bit instruction**.
- Load word to register using a pointer register with 16-bit offset for the address or using a pointer register offset by a second pointer for the address encodes as a **32-bit instruction**.
- Load word to register using a 32-bit absolute address encodes as a **64-bit instruction**.

The 16-bit load word instructions may be issued in parallel with certain other instructions. The 32- and 64-bit load word instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdM16bitToDreg Example

```
r3 = w [ p0 ] (z) ;
r7 = w [ p1 ++ ] (z) ;
r2 = w [ sp -- ] (z) ;
r6 = w [ p2 + 12 ] (z) ;
r0 = w [ p4 + 0x8004 ] (z) ;
r1 = w [ p0 ++ p1 ] (z) ;
r3 = w [ p0 ] (x) ;
r7 = w [ p1 ++ ] (x) ;
r2 = w [ sp -- ] (x) ;
r6 = w [ p2 + 12 ] (x) ;
r0 = w [ p4 + 0x800E ] (x) ;
r1 = w [ p0 ++ p1 ] (x) ;
```

16-Bit Load from Memory (LdM16bitToDregH)

General Form

LdStPmod
DREG_H = w[PREG ++ PREG]Missing syntax here: Dreg_hi = w [Preg]
DspLdSt
DREG_H = w[IREG ++]
DREG_H = w[IREG --]
DREG_H = w[IREG]
LdStAbs
DREG_H = w[uimm32]

Abstract

This instruction loads a high-half register with a 16-bit value from memory.

See Also (LdM08bitToDreg, LdM32bitToDreg, LdM16bitToDregL, LdM16bitToDreg)

LdM16bitToDregH Description

The load word to high-half data register instruction loads a 16-bit value from a memory location into a 16-bit high-half data register. The operation does not affect the related low-half register. The address of the memory location is identified with an index register, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (2-byte half-word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 2 bytes [*Ireg* ++]
- Post-decrement the source index by 2 bytes [*Ireg* --]
- Offset the source pointer with second pointer [*Preg* ++ *Preg*]

The instruction versions that explicitly modify an index register (*Ireg*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*Lreg*) corresponding to the *Ireg* used in this instruction. For example, if you use I2 to increment your address pointer, first clear L2 to disable circular buffering. Failure to explicitly clear *Lreg* beforehand can result in unexpected *Ireg* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the

circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
Dest_hi = w [ Src_1 ++ Src_2 ] ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dest_hi = w [Src_1] ; /* load the 16-bit destination, indirect*/
Src_1 += Src_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- *Dest_hi* is the destination high-half register. (*Dreg_hi* in the syntax example).
- *Src_1* is the first source register on the right-hand side of the equation.
- *Src_2* is the second source register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
r0.h = w [p2 ++ p2] ;
```

functions as:

```
r0.h = w [p2] ;
```

The instruction opcode size varies with the address type as follows:

- Load word to high-half register using an index register or a pointer register for the address encodes as a **16-bit instruction**.
- Load word to high-half register using a pointer register offset by a second pointer for the address encodes as a **16-bit instruction**.
- Load word to high-half register using a 32-bit absolute address encodes as a **64-bit instruction**.

The 16-bit load word instructions may be issued in parallel with certain other instructions. The 64-bit load word instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdM16bitToDregH Example

```
r3.h = w [ i1 ] ;
r7.h = w [ i3 ++ ] ;
r1.h = w [ i0 -- ] ;
r2.h = w [ p4 ] ;
r5.h = w [ p2 ++ p0 ] ;
```

16-Bit Load from Memory (LdM16bitToDregL)

General Form

LdStPmod
DREG_L = w[PREG ++ PREG]Missing syntax here: Dreg_lo = w [Preg]
DspLdSt
DREG_L = w[IREG ++]
DREG_L = w[IREG --]
DREG_L = w[IREG]
LdStAbs
DREG_L = w[uimm32]

Abstract

This instruction loads a low-half register with a 16-bit value from memory.

See Also (LdM08bitToDreg, LdM32bitToDreg, LdM16bitToDregH, LdM16bitToDreg)

LdM16bitToDregL Description

The load word to low-half data register instruction loads a 16-bit value from a memory location into a 16-bit low-half data register. The operation does not affect the related high-half register. The address of the memory location is identified with an index register, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (2-byte half-word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 2 bytes [*Ireg* ++]
- Post-decrement the source index by 2 bytes [*Ireg* --]
- Offset the source pointer with second pointer [*Preg* ++ *Preg*]

The instruction versions that explicitly modify an index register (*Ireg*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*Lreg*) corresponding to the *Ireg* used in this instruction. For example, if you use I2 to increment your address pointer, first clear L2 to disable circular buffering. Failure to explicitly clear *Lreg* beforehand can result in unexpected *Ireg* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the

circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
Dest_lo = w [ Src_1 ++ Src_2 ] ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dest_lo = w [Src_1] ; /* load the 16-bit destination, indirect*/
Src_1 += Src_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- *Dest_lo* is the destination low-half register. (*Dreg_lo* in the syntax example).
- *Src_1* is the first source register on the right-hand side of the equation.
- *Src_2* is the second source register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
r0.l = w [p2 ++ p2] ;
```

functions as:

```
r0.l = w [p2] ;
```

The instruction opcode size varies with the address type as follows:

- Load word to low-half register using an index register or a pointer register for the address encodes as a **16-bit instruction**.
- Load word to low-half register using a pointer register offset by a second pointer for the address encodes as a **16-bit instruction**.
- Load word to low-half register using a 32-bit absolute address encodes as a **64-bit instruction**.

The 16-bit load word instructions may be issued in parallel with certain other instructions. The 64-bit load word instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdM16bitToDregL Example

```
r3.l = w[ i1 ] ;
r7.l = w[ i3 ++ ] ;
r1.l = w[ i0 -- ] ;
r2.l = w[ p4 ] ;
r5.l = w[ p2 ++ p0 ] ;
```

32-Bit Load from Memory (LdM32bitToDreg)

General Form

LdStPmod
DREG =[PREG ++ PREG]
DspLdSt
DREG =[IREG ++]
DREG =[IREG --]
DREG =[IREG]
DREG =[IREG ++ MREG]
LdSt
DREG =[PREG ++]
DREG =[PREG --]
DREG =[PREG]
LdStIIFP
DREG =[fp - imm5nzs4negpos]
LdpIIFP
PREG =[fp - imm5nzs4negpos]
LdStII
DREG =[PREG + uimm4s4]
LdStIdxI
DREG =[PREG + imm16s4]
LdStAbs
DREG =[uimm32]
PREG =[uimm32]

Abstract

This instruction loads a register with a 32-bit value from memory.

See Also (LdM08bitToDreg, LdM16bitToDregH, LdM16bitToDregL, LdM16bitToDreg)

LdM32bitToDreg Description

The load 32-bit data to data register instruction loads a 32-bit value from a memory location into a data register. The address of the memory location is identified with an index register, an index register plus an offset, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (4-byte word address alignment). Failure to

maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 4 bytes [*Ireg* ++]
- Post-decrement the source index by 4 bytes [*Ireg* --]
- Offset the source index with a modifier [*Ireg* ++ *Mreg*]
- Post-increment the source pointer by 4 bytes [*Preg* ++]
- Post-decrement the source pointer by 4 bytes [*Preg* --]
- Offset the source frame pointer with a 5-bit signed constant [FP - *SmallOffset*]
- Offset the source pointer with a 5-bit signed constant [*Preg* + *SmallOffset*]
- Offset the source pointer with a 16-bit signed constant [*Preg* + *LargeOffset*]
- Offset the source pointer with second pointer [*Preg* ++ *Preg*]

The instruction versions that explicitly modify an index register (*Ireg*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*Lreg*) corresponding to the *Ireg* used in this instruction. For example, if you use I2 to increment your address pointer, first clear L2 to disable circular buffering. Failure to explicitly clear *Lreg* beforehand can result in unexpected *Ireg* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
Dest = [ Src_1 ++ Src_2 ] ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dest = [Src_1] ; /* load the 32-bit destination, indirect*/
Src_1 += Src_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- *Dest* is the destination high-half register. (*Dreg* in the syntax example).
- *Src_1* is the first source register on the right-hand side of the equation.
- *Src_2* is the second source register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
r0 = [p2 ++ p2] ;
```

functions as:

```
r0 = [p2] ;
```

The instruction opcode size varies with the address type as follows:

- Load 32-bit data to register using an index register or a pointer register for the address encodes as a **16-bit instruction**.
- Load 32-bit data to register using an index register offset by a modifier register or a pointer register offset by a second pointer for the address encodes as a **16-bit instruction**.
- Load 32-bit data to register using a pointer or frame pointer register with a small offset for the address encodes as a **16-bit instruction**.
- Load 32-bit data to register using a pointer register with a large offset for the address encodes as a **32-bit instruction**.
- Load 32-bit data to register using a 32-bit absolute address encodes as a **64-bit instruction**.

The 16-bit load 32-bit data to register instructions may be issued in parallel with certain other instructions. The 32- and 64-bit load 32-bit data to register instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdM32bitToDreg Example

```
r3 = [ p0 ] ;
r7 = [ p1 ++ ] ;
r2 = [ sp -- ] ;
r6 = [ p2 + 12 ] ;
r0 = [ p4 + 0x800C ] ;
r1 = [ p0 ++ p1 ] ;
r5 = [ fp -12 ] ;
r2 = [ i2 ] ;
r0 = [ i0 ++ ] ;
r0 = [ i0 -- ] ;
/* Before indirect post-increment indexed addressing*/
r7 = 0 ;
i3 = 0x4000 ; /* Memory location contains 15, for example.*/
m0 = 4 ;
r7 = [i3 ++ m0] ;
  /* Afterwards . . .*/
  /* r7 = 15 from memory location 0x4000*/
```

```
/* i3 = i3 + m0 = 0x4004*/
/* m0 still equals 4*/
```

32-Bit Pointer Load from Memory (LdM32bitTo-Preg)

General Form

Ldp
PREG =[PREG ++]
PREG =[PREG --]
PREG =[PREG]
LdpII
PREG =[PREG + uimm4s4]
LdStIdxI
PREG =[PREG + imm16s4]

Abstract

This instruction loads a pointer register with 7-bit immediate value.

LdM32bitToPreg Description

The load 32-bit data to pointer register instruction loads a 32-bit value from a memory location into a pointer register. The address of the memory location is identified with a pointer register or a pointer plus an offset. The address used in this instruction is restricted to even memory address alignment (4-byte word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source pointer by 4 bytes `[Preg ++]`
- Post-decrement the source pointer by 4 bytes `[Preg --]`
- Offset the source pointer with a 5-bit signed constant `[Preg + SmallOffset]`
- Offset the source pointer with a 16-bit signed constant `[Preg + LargeOffset]`

The instruction opcode size varies with the address type as follows:

- Load 32-bit data to register using a pointer register for the address encodes as a **16-bit instruction**.
- Load 32-bit data to register using a pointer register with a small offset for the address encodes as a **16-bit instruction**.

- Load 32-bit data to register using a pointer register with a large offset for the address encodes as a **32-bit instruction**.

The 16-bit load 32-bit data to pointer instructions may be issued in parallel with certain other instructions. The 32-bit load 32-bit data to pointer instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

LdM32bitToPreg Example

```
p3 = [ p2 ] ;
p5 = [ p0 ++ ] ;
p2 = [ sp -- ] ;
p3 = [ p2 + 8 ] ;
p0 = [ p2 + 0x4008 ] ;
p1 = [ fp - 16 ] ;
```

Memory Load (Exclusive) Operations

The memory load (exclusive) operations read data from memory (similar to a non-exclusive, "regular" memory load operation) and establish exclusive access to the memory location. When the memory location is in *non-shareable* memory, the memory load (exclusive) operation loads through the memory management unit in exactly the same manner as a regular memory load from the same memory location. When the memory location is in *shareable* memory, the memory load (exclusive) performs an exclusive read on the memory bus. For more information about illegal, non-shareable, and shareable memory regions, see the Exclusive Loads and Stores section of the Memory chapter.

The memory load (exclusive) operations include the following instructions:

- 8-Bit Load from Memory to 32-bit Register (LdX08bitToDreg)
- 32-Bit Load from Memory (LdX32bitToDreg)
- 16-Bit Load from Memory (LdX16bitToDregH)
- 16-Bit Load from Memory (LdX16bitToDregL)
- 16-Bit Load from Memory to 32-Bit Register (LdX16bitToDreg)

When the memory management unit successfully completes an memory load (exclusive), the destination data register is updated with the loaded value and the `SEQSTAT.XMONITOR` bit is set, as shown in the **Exclusive Related Bits in Status Register (SEQSTAT)** table.

Table 8-39: Exclusive Related Bits in Status Register (SEQSTAT)

Name	Description	Condition
`SEQSTAT.XMONITOR`	Excl. monitor (0=open, 1=exclusive)	Always updated
	=1 after completion of exclusive load	

When the memory management unit cannot complete an memory load (exclusive) a number of exceptions and errors may be issued, in addition to those that may be caused by a regular memory load. The **Exceptions/Errors from Unsuccessful Memory Load (Exclusive) Operations** table lists these exceptions and errors.

Table 8-40: Exceptions/Errors from Unsuccessful Memory Load (Exclusive) Operations

Condition	Exception or Hardware Error
Access to misaligned address	The data access generated a misaligned address violation exception. The address for the exclusive access must be aligned. This restriction holds even if misaligned accesses are supported generally.
Access to core MMR	The data access attempted an illegal use of a supervisor resource.
Access to I/O device space	The data access generated a CPLB protection violation exception. This exception occurs when the access is to memory marked as I/O device space in the CPLB.

Table 8-40: Exceptions/Errors from Unsuccessful Memory Load (Exclusive) Operations (Continued)

Condition	Exception or Hardware Error
Access to non-exclusive slave	A load exclusive from a shareable memory region accessed a memory device without hardware support for exclusive accesses. The program which receives this exception should not execute the store exclusive to this address. The store exclusive will be unsuccessful causing the exclusive instruction sequence to be retried and the program to loop. This condition has to be treated as a programming error and the program restructured to place the semaphore in another memory location, or use the TESTSET instruction.

8-Bit Load from Memory to 32-bit Register (LdX08bitToDreg)

General Form

LdStExcl
DREG = b[PREG] (z,excl)
DREG = b[PREG] (x,excl)

Abstract

This instruction loads a register with an 8-bit value from memory, using an exclusive memory read. The value is sign or zero extended in the register.

See Also (LdX32bitToDreg, LdX16bitToDregH, LdX16bitToDregL, LdX16bitToDreg)

LdX08bitToDreg Description

The load data register from memory (8-bit transfer) checks for exclusive access to a memory location and reads a data value (loading it into the least significant byte of the register with zero- or sign-extension) from the location only if exclusive access is still held.

For more information about memory load (exclusive) operations, see Memory Load (Exclusive) Operations.

LdX08bitToDreg Example

```
r1 = b[p4] (x,excl);   /* load exclusive 8-bits sign sign to D-register */
```

16-Bit Load from Memory to 32-Bit Register (LdX16bitToDreg)

General Form

LdStExcl
DREG = w[PREG] (z,excl)
DREG = w[PREG] (x,excl)

Abstract

This instruction loads a register with a 16-bit value from memory, using an exclusive memory read. The value is sign or zero extended in the register.

See Also (LdX08bitToDreg, LdX32bitToDreg, LdX16bitToDregH, LdX16bitToDregL)

LdX16bitToDreg Description

The load data register from memory (16-bit transfer) checks for exclusive access to a memory location and reads a data value (loading it into the least significant 16-bits of the register with zero- or sign-extension) from the location only if exclusive access is still held.

For more information about memory load (exclusive) operations, see Memory Load (Exclusive) Operations.

LdX16bitToDreg Example

```
r2 = w[p4] (z,excl);   /* load exclusive 16-bits zero extend to D-register */
r3 = w[p4] (x,exc;)   /* load exclusive 16-bits sign extend to D-register */
```

16-Bit Load from Memory (LdX16bitToDregH)

General Form

LdStExcl
DREG_H = w[PREG] (excl)

Abstract

This instruction loads a high-half register with a 16-bit value from memory, using an exclusive memory read.

See Also (LdX08bitToDreg, LdX32bitToDreg, LdX16bitToDregL, LdX16bitToDreg)

LdX16bitToDregH Description

The load high half data register from memory (16-bit transfer) checks for exclusive access to a memory location and reads a data value (loading it into the half register) from the location only if exclusive access is still held.

For more information about memory load (exclusive) operations, see Memory Load (Exclusive) Operations.

LdX16bitToDregH Example

```
r1.h = w[p4] (excl);   /* load exclusive 16-bits zero sign to high D-register half */
```

16-Bit Load from Memory (LdX16bitToDregL)

General Form

LdStExcl
DREG_L = w[PREG] (excl)

Abstract

This instruction loads a low-half register with a 16-bit value from memory, using an exclusive memory read.

See Also (LdX08bitToDreg, LdX32bitToDreg, LdX16bitToDregH, LdX16bitToDreg)

LdX16bitToDregL Description

The load low half data register from memory (16-bit transfer) checks for exclusive access to a memory location and reads a data value (loading it into the half register) from the location only if exclusive access is still held.

For more information about memory load (exclusive) operations, see Memory Load (Exclusive) Operations.

LdX16bitToDregL Example

```
r1.l = w[p4] (excl);   /* load exclusive 16-bits zero sign to low D-register half */
```

32-Bit Load from Memory (LdX32bitToDreg)

General Form

LdStExcl
DREG = [PREG] (excl)

Abstract

This instruction loads a register with a 32-bit value from memory, using an exclusive memory read.

See Also (LdX08bitToDreg, LdX16bitToDregH, LdX16bitToDregL, LdX16bitToDreg)

LdX32bitToDreg Description

The load data register from memory (32-bit transfer) checks for exclusive access to a memory location and reads a data value (loading it into the register) from the location only if exclusive access is still held.

For more information about memory load (exclusive) operations, see Memory Load (Exclusive) Operations.

LdX32bitToDreg Example

```
r0 = [p4] (excl);      /* load exclusive 32-bits to D-register */
```

Pack Operations

These operations provide byte packing and unpacking operations on register and register pair operands:

- Pack 8-Bit to 32-Bit (BytePack)
- Spread 8-Bit to 16-Bit (ByteUnPack)
- Pack 16-Bit to 32-Bit (Pack16Vec)

Pack 8-Bit to 32-Bit (BytePack)

General Form

Dsp32Alu
DREG = bytepack (DREG , DREG)

Abstract

This instruction takes the low bytes from each 16-bit register half of two registers and combines them to create a single 32-bit register. Used to re-order data.

See Also (ByteUnPack, Pack16Vec)

BytePack Description

The Quad 8-Bit Pack instruction packs four 8-bit values, half-word aligned, contained in two source registers into one register, byte aligned. The **Source Registers Contain** figure and **Destination Register Receives** figure show the packing pattern.

Figure 8-21: Source Registers Contain

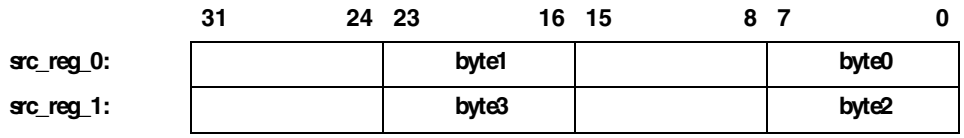
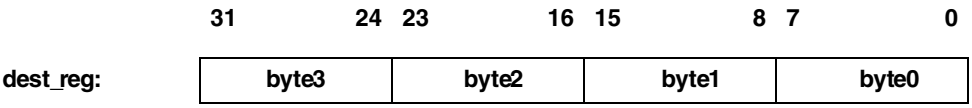


Figure 8-22: Destination Register Receives



This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

This **16-bit instruction** may be issued in parallel with other 16-bit instructions.

This instruction may be used in either **User or Supervisor mode**.

BytePack Example

```
r2 = bytepack (r4,r5) ;
/* Assume the following: ... */
/*    R4 = 0xFEED FACE */
/*    R5 = 0xBEEF BADD */
```

```
/* Then, this instruction returns: ... */
/*    R2 = 0xEFDD EDCE */
```

Spread 8-Bit to 16-Bit (ByteUnPack)

General Form

Dsp32Alu
(DREG , DREG) = byteunpack PAIR0 RS

Abstract

This instruction spreads four bytes to four zero extended 16-Bit values. The lower two bits of I0 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair.

See Also (BytePack, Pack16Vec)

ByteUnPack Description

The Quad 8-Bit Unpack instruction copies four contiguous bytes from a pair of source registers, adjusting for byte alignment. The instruction loads the selected bytes into two arbitrary data registers on half-word alignment. The two LSBs of the I0 register determine the source byte alignment, as shown in the **I-register Bits and the Byte Alignment, no (r) option** figure. This figure shows the default source order case---not the (r) syntax---and the data contained in the source register pair. This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

Figure 8-23: I-register Bits and the Byte Alignment, no (r) option

The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

The (r) syntax reverses the order of the source registers within the pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (r) option causes the low order bytes to come from the high register. In the optional reverse source order case (for example, using the (r) syntax), the only

difference is the source registers swap places in their byte ordering. Assume the source register pair contains the data shown in the **I-register Bits and the Byte Alignment, with (r) option** figure.

Figure 8-24: I-register Bits and the Byte Alignment, with (r) option

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

The four bytes, now byte aligned, are copied into the destination registers on half-word alignment, as shown in the **Source Register Contains** figure and the **Destination Registers Receive** figure.

Figure 8-25: Source Register Contains

	31 24	23 16	15 8	7 0
Aligned bytes :	byte_D	byte_C	byte_B	byte_A

Figure 8-26: Destination Registers Receive

	31 24	23 16	15 8	7 0
dest_reg_0:		byte_B		byte_A
dest_reg_1:		byte_D		byte_C

Only register pairs `R1:0` and `R3:2` are valid sources for this instruction, and the destination registers must be unique. Misaligned access exceptions are disabled during this instruction.

This **16-bit instruction** may be issued in parallel with other 16-bit instructions.

This instruction may be used in either **User or Supervisor mode**.

ByteUnPack Example

```
(r6,r5) = byteunpack r1:0 ; /* non-reversing sources */
/* Assume the following: */
/* ... register I0's two LSBs = 00b,  */
/* ... R1 = 0xFEED FACE  */
/* ... R0 = 0xBEEF BADD  */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00BE 00EF  */
/* ... R5 = 0x00BA 00DD  */
/* Assume the following: */
/* ... register I0's two LSBs = 01b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
```

```
/* ... Then, this instruction returns: */
/* ... R6 = 0x00CE 00BE */
/* ... R5 = 0x00EF 00BA */
/* Assume the following: */
/* ... register I0's two LSBs = 10b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00FA 00CE */
/* ... R5 = 0x00BE 00EF */
/* Assume the following: */
/* ... register I0's two LSBs = 11b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00ED 00FA */
/* ... R5 = 0x00CE 00BE */


(r6,r5) = byteunpack r1:0 (R) ; /* reversing sources case */
/* Assume the following: */
/* ... register I0's two LSBs = 00b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00FE 00ED */
/* ... R5 = 0x00FA 00CE */
/* Assume the following: */
/* ... register I0's two LSBs = 01b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00DD 00FE */
/* ... R5 = 0x00ED 00FA */
/* Assume the following: */
/* ... register I0's two LSBs = 10b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00BA 00DD */
/* ... R5 = 0x00FE 00ED */
/* Assume the following: */
/* ... register I0's two LSBs = 11b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00EF 00BA */
/* ... R5 = 0x00DD 00FE */
```

Pack 16-Bit to 32-Bit (Pack16Vec)

General Form

Dsp32Shf
DREG = pack (DREG_L , DREG_L)
DREG = pack (DREG_L , DREG_H)
DREG = pack (DREG_H , DREG_L)
DREG = pack (DREG_H , DREG_H)

Abstract

This instruction packs two 16-bit half registers into one 32-bit register.

See Also (BytePack, ByteUnPack)

Pack16Vec Description

The vector pack instruction packs two 16-bit half-word numbers into the halves of a 32-bit data register as shown in the **Source Registers Contain** figure and the **Destination Register Contains** figure.

Figure 8-27: Source Registers Contain

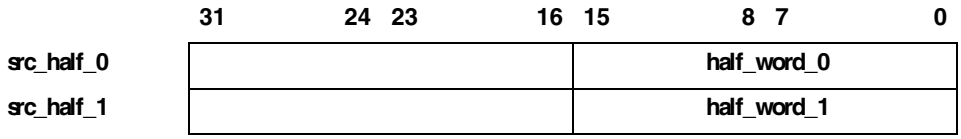
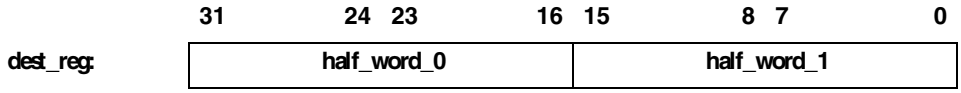


Figure 8-28: Destination Register Contains



This **16-bit instruction** may be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either **User or Supervisor mode**.

Pack16Vec Example

```
r3=pack(r4.l, r5.l) ; /* pack low / low half-words */
r1=pack(r6.l, r4.h) ; /* pack low / high half-words */
r0=pack(r2.h, r4.l) ; /* pack high / low half-words */
r5=pack(r7.h, r2.h) ; /* pack high / high half-words */
```

```
/* Special Applications */
/* If r4.l = 0xDEAD and r5.l = 0xBEEF, then . . . */
r3 = pack (r4.l, r5.l) ;
/* . . . produces r3 = 0xDEAD BEEF */
/* example needed here */
```

Memory Store Operations

These operations provide memory store operations on register and immediate value operands:

- 8-Bit Store to Memory (StDregToM08bit)
- 16-Bit Store to Memory (StDregLToM16bit)
- 16-Bit Store to Memory (StDregHToM16bit)
- 32-Bit Store to Memory (StDregToM32bit)
- Store Pointer (StPregToM32bit)

16-Bit Store to Memory (StDregHToM16bit)

General Form

LdStPmod
w[PREG ++ PREG] = DREG_H *Missing syntax here: w [Preg] = Dreg_hi*
DspLdSt
w[IREG ++] = DREG_H
w[IREG --] = DREG_H
w[IREG] = DREG_H
LdStAbs
w[uimm32] = DREG_H

Abstract

This instruction stores the most significant 16-bit value from a register to memory.

See Also (StDregToM08bit, StDregToM32bit, StDregLToM16bit)

StDregHToM16bit Description

The store word from high-half data register instruction stores a 16-bit value from a high-half data register to a memory location. The operation does not affect the related low-half register. The address of the memory location is identified with an index register, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (2-byte half-word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 2 bytes [*Ireg* ++]
- Post-decrement the source index by 2 bytes [*Ireg* --]
- Offset the source pointer with second pointer [*Preg* ++ *Preg*]

The instruction versions that explicitly modify an index register (*Ireg*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*Lreg*) corresponding to the *Ireg* used in this instruction. For example, if you use I2 to increment your address pointer, first clear L2 to disable circular buffering. Failure to explicitly clear *Lreg* beforehand can result in unexpected *Ireg* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the

circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
w [ Dest_1 ++ Dest_2 ] = Src_hi ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
w [Dest_1] = Src_hi ; /* store to the 16-bit destination, indirect*/
Dest_1 += Dest_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- *Src_hi* is the source high-half register. (*Dreg_hi* in the syntax example).
- *Dest_1* is the first destination register on the left-hand side of the equation.
- *Dest_2* is the second destination register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
w [p2 ++ p2] = r0.h ;
```

functions as:

```
w [p2] = r0.h ;
```

The instruction opcode size varies with the address type as follows:

- Store word to memory using an index register or a pointer register for the address encodes as a **16-bit instruction**.
- Store word to memory using a pointer register offset by a second pointer for the address encodes as a **16-bit instruction**.
- Store word to memory using a 32-bit absolute address encodes as a **64-bit instruction**.

The 16-bit load word instructions may be issued in parallel with certain other instructions. The 64-bit load word instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

StDregHToM16bit Example

```
w[ i1 ] = r3.h ;
w[ i3 ++ ] = r7.h ;
w[ i0 -- ] = r1.h ;
w[ p4 ] = r2.h ;
w[ p2 ++ p0 ] = r5.h ;
```

16-Bit Store to Memory (StDregLToM16bit)

General Form

LdStPmod
w[PREG ++ PREG] = DREG_L
DspLdSt
w[IREG ++] = DREG_L
w[IREG --] = DREG_L
w[IREG] = DREG_L
LdSt
w[PREG ++] = DREG
w[PREG --] = DREG
w[PREG] = DREG
LdStII
w[PREG + uimm4s2] = DREG
LdStIdxI
w[PREG + imm16s2] = DREG
LdStAbs
w[uimm32] = DREG

Abstract

This instruction stores the least significant 16-bit value from a register to memory.

See Also (StDregToM08bit, StDregToM32bit, StDregHToM16bit)

StDregLToM16bit Description

The store word from low-half data register instruction stores a 16-bit value *either* from a low-half data register *or* from the least significant 16 bits of a data register to a memory location. The operation does not affect the related high-half register. The address of the memory location is identified with an index register, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (2-byte half-word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 2 bytes [*Ireg* ++]
- Post-decrement the source index by 2 bytes [*Ireg* --]

- Offset the source pointer with second pointer `[Preg ++ Preg]`

The instruction versions that explicitly modify an index register (*Ireg*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*Lreg*) corresponding to the *Ireg* used in this instruction. For example, if you use `I2` to increment your address pointer, first clear `L2` to disable circular buffering. Failure to explicitly clear *Lreg* beforehand can result in unexpected *Ireg* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
w [ Dest_1 ++ Dest_2 ] = Src_hi ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
w [Dest_1] = Src_hi ; /* store to the 16-bit destination, indirect*/
Dest_1 += Dest_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- *Src_hi* is the source high-half register. (*Dreg_hi* in the syntax example).
- *Dest_1* is the first destination register on the left-hand side of the equation.
- *Dest_2* is the second destination register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
w [p2 ++ p2] = r0.h ;
```

functions as:

```
w [p2] = r0.h ;
```

The instruction opcode size varies with the address type as follows:

- Store word to memory using an index register or a pointer register for the address encodes as a **16-bit instruction**.
- Store word to memory using a pointer register offset by a second pointer for the address encodes as a **16-bit instruction**.
- Store word to memory using a 32-bit absolute address encodes as a **64-bit instruction**.

The 16-bit load word instructions may be issued in parallel with certain other instructions. The 64-bit load word instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

StDregLToM16bit Example

```
w [ i1 ] = r3.l ;
w [ p0 ] = r3 ;
w [ i3 ++ ] = r7.l ;
w [ i0 -- ] = r1.l ;
w [ p4 ] = r2.l ;
w [ p1 ++ ] = r7 ;
w [ sp -- ] = r2 ;
w [ p2 + 12 ] = r6 ;
w [ p4 - 0x200C ] = r0 ;
w [ p2 ++ p0 ] = r5.l ;
```

8-Bit Store to Memory (StDregToM08bit)

General Form

LdSt
b[PREG ++] = DREG
b[PREG --] = DREG
b[PREG] = DREG
LdStIdxI
b[PREG + imm16reloc] = DREG
LdStAbs
b[uimm32] = DREG

Abstract

This instruction stores the least significant 8-bit value from a register to memory.

See Also (StDregToM32bit, StDregLToM16bit, StDregHToM16bit)

StDregToM08bit Description

The store byte from data register instruction stores the least significant 8 bits from a 32-bit data register byte to a memory location. The address of the memory location is identified with a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction has no restrictions for memory address alignment. This instruction supports the following options.

- Post-increment the source pointer by 1 byte [*Preg ++*]
- Post-decrement the source pointer by 1 byte [*Preg --*]
- Offset the source pointer with a 16-bit signed constant [*Preg + Offset*]

The instruction opcode size varies with the address type as follows:

- Store byte to memory using a pointer register for the address encodes as a **16-bit instruction**.
- Store byte to memory using a pointer register with 16-bit offset for the address encodes as a **32-bit instruction**.
- Store byte to memory using a 32-bit absolute address encodes as a **64-bit instruction**.

The 16-bit store byte instructions may be issued in parallel with certain other instructions. The 32- and 64-bit load byte instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

StDregToM08bit Example

```
b [ p0 ] = r3 ;
b [ p1 ++ ] = r7 ;
b [ sp -- ] = r2 ;
b [ p4 + 0x100F ] = r0 ;
b [ p4 - 0x53F ] = r0 ;
```

32-Bit Store to Memory (StDregToM32bit)

General Form

LdStPmod
[PREG ++ PREG] = DREG
DspLdSt
[IREG ++] = DREG
[IREG --] = DREG
[IREG] = DREG
[IREG ++ MREG] = DREG
LdSt
[PREG ++] = DREG
[PREG --] = DREG
[PREG] = DREG
LdStIIFP
[fp - imm5nzs4negpos] = DREG
[fp - imm5nzs4negpos] = PREG
LdStII
[PREG + uimm4s4] = DREG
LdStIdxI
[PREG + imm16s4] = DREG
LdStAbs
[uimm32] = DREG
[uimm32] = PREG

Abstract

This instruction stores the 32-bit value from a register to memory.

See Also (StDregToM08bit, StDregLToM16bit, StDregHToM16bit)

StDregToM32bit Description

The store 32-bit data from data register instruction stores a 32-bit value from a data register into a memory location. The address of the memory location is identified with an index register, an index register plus an offset, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (4-byte word address alignment). Failure to

maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 4 bytes `[Ireg ++]`
- Post-decrement the source index by 4 bytes `[Ireg --]`
- Offset the source index with a modifier `[Ireg ++ Mreg]`
- Post-increment the source pointer by 4 bytes `[Preg ++]`
- Post-decrement the source pointer by 4 bytes `[Preg --]`
- Offset the source frame pointer with a 5-bit signed constant `[FP - SmallOffset]`
- Offset the source pointer with a 5-bit signed constant `[Preg + SmallOffset]`
- Offset the source pointer with a 16-bit signed constant `[Preg + LargeOffset]`
- Offset the source pointer with second pointer `[Preg ++ Preg]`

The instruction versions that explicitly modify an index register (*Ireg*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*Lreg*) corresponding to the *Ireg* used in this instruction. For example, if you use `I2` to increment your address pointer, first clear `L2` to disable circular buffering. Failure to explicitly clear *Lreg* beforehand can result in unexpected *Ireg* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
[ Dest_1 ++ Dest_2 ] = Src ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
[Dest_1] = Src ; /* store to the 32-bit destination, indirect*/
Dest_1 += Dest_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- *Src* is the source register. (*Dreg* in the syntax example).
- *Dest_1* is the first destination register on the right-hand side of the equation.
- *Dest_2* is the second destination register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
[p2 ++ p2] = r0 ;
```

functions as:

```
[p2] = r0 ;
```

The instruction opcode size varies with the address type as follows:

- Store 32-bit data to memory using an index register or a pointer register for the address encodes as a **16-bit instruction**.
- Store 32-bit data to memory using an index register offset by a modifier register or a pointer register offset by a second pointer for the address encodes as a **16-bit instruction**.
- Store 32-bit data to memory using a pointer or frame pointer register with a small offset for the address encodes as a **16-bit instruction**.
- Store 32-bit data to memory using a pointer register with a large offset for the address encodes as a **32-bit instruction**.
- Store 32-bit data to memory using a 32-bit absolute address encodes as a **64-bit instruction**.

The 16-bit store 32-bit data to register instructions may be issued in parallel with certain other instructions. The 32- and 64-bit store 32-bit data to register instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

StDregToM32bit Example

```
[ p0 ] = r3 ;
[ p1 ++ ] = r7 ;
[ sp -- ] = r2 ;
[ p2 + 12 ] = r6 ;
[ p4 - 0x1004 ] = r0 ;
[ p0 ++ p1 ] = r1 ;
[ fp - 28 ] = r5 ;
[ i2 ] = r2 ;
[ i0 ++ ] = r0 ;
[ i0 -- ] = r0 ;
[ i3 ++ m0 ] = r7 ;
```

Store Pointer (StPregToM32bit)

General Form

LdSt
[PREG ++] = PREG
[PREG --] = PREG
[PREG] = PREG
LdStII
[PREG + uimm4s4] = PREG
LdStIdxI
[PREG + imm16s4] = PREG

Abstract

This instruction stores the 32-bit value from a pointer register to memory.

StPregToM32bit Description

The store 32-bit data from pointer register instruction stores a 32-bit value from a pointer register into a memory location. The address of the memory location is identified with a pointer register or a pointer plus an offset. The address used in this instruction is restricted to even memory address alignment (4-byte word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source pointer by 4 bytes `[Preg ++]`
- Post-decrement the source pointer by 4 bytes `[Preg --]`
- Offset the source pointer with a 5-bit signed constant `[Preg + SmallOffset]`
- Offset the source pointer with a 16-bit signed constant `[Preg + LargeOffset]`

The instruction opcode size varies with the address type as follows:

- Store 32-bit data to memory using a pointer register for the address encodes as a **16-bit instruction**.
- Store 32-bit data to memory using a pointer register with a small offset for the address encodes as a **16-bit instruction**.
- Store 32-bit data to memory using a pointer register with a large offset for the address encodes as a **32-bit instruction**.

The 16-bit store 32-bit data from pointer instructions may be issued in parallel with certain other instructions. The 32-bit store 32-bit data from pointer instructions may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

StPregToM32bit Example

```
[ p2 ] = p3 ;
[ sp ++ ] = p5 ;
[ p0 -- ] = p2 ;
[ p2 + 8 ] = p3 ;
[ p2 + 0x4444 ] = p0 ;
[ fp -12 ] = p1 ;
```

Memory Store (Exclusive) Operations

The memory store (exclusive) instruction forms the second part of the exclusive instruction sequence, in which the instruction has to establish exclusive access to a memory location. The store exclusive instruction only modifies memory if the current task (for example, core or thread) still has exclusive access to that location. An intervening load exclusive by another task, a write to the memory location, or a SYNCEXCL instruction may have caused the exclusive access to have been lost.

Checking for exclusive access is a two stage process. First the SEQSTAT.XMONITOR bit is tested. If this is 0 the store exclusive instruction terminates immediately. If SEQSTAT.XMONITOR is 1, the store exclusive instruction attempts to write data to the memory location. When the memory location is in *non-shareable* memory, the instruction is considered to have exclusive access to the location simply based on the SEQSTAT.XMONITOR bit, and the store exclusive instruction performs the write in exactly the same manner as a regular memory store to the same memory location. When the memory location is in *shareable* memory, the store exclusive instruction performs the write with an exclusive transaction on the memory bus. This transaction may itself fail to update the location if another core has established exclusive access or written to the location since the current task executed the prior load exclusive instruction. For more information about illegal, non-shareable, and shareable memory regions, see the Exclusive Loads and Stores section of the Memory chapter.

The memory store (exclusive) operations include the following instructions:

- 8-Bit Store to Memory (StDregToX08bit)
- 32-Bit Store to Memory (StDregToX32bit)
- 16-Bit Store to Memory (StDregLToX16bit)
- 16-Bit Store to Memory (StDregHToX16bit)

The store exclusive instruction terminates before the success or failure of the write transaction is known. The state of the write transaction is tracked in the SEQSTAT.XWACTIVE and SEQSTAT.XWAVAIL bits which are updated asynchronously to the core pipeline once the write response has been received from the system. These bits should not be tested directly, instead the SYNCEXCL (Synchronize Exclusive State) instruction should be used to waits for the write to complete and set ASTAT.CC according to whether the store exclusive instruction successfully wrote to the memory location. The **Exclusive Related Bits in Status Registers (SEQSTAT and ASTAT)** table shows how exclusive access status changes during an exclusive memory store operation.

Table 8-41: Exclusive Related Bits in Status Registers (SEQSTAT and ASTAT)

Name	Description	Condition
SEQSTAT.XMONITOR	Exclusive monitor (0=open, 1=exclusive)	Not updated
	=0 on start of instruction, CC=0	
	=1 on start of instruction, attempt update (Preg,val), ASTAT.CC=1, SEQSTAT.XWACTIVE=1	

Table 8-41: Exclusive Related Bits in Status Registers (SEQSTAT and ASTAT) (Continued)

Name	Description	Condition
SEQSTAT.XWACTIVE	Exclusive write active (0=no status[1], 1=active)	Always updated
	=0 on completion of instruction	
	=1 while active	
ASTAT.CC	Condition Code (0=no write attempted, 1=write attempted)	Always updated
SEQSTAT.XWAVAIL	Exclusive write resp. (0=no status, 1=available)	Always updated
	=1 on completion of write transaction	

1. If XWACTIVE is not 0 when the instruction starts, the instruction throws an exception.

When the memory management unit cannot complete an memory store (exclusive) a number of exceptions and errors may be issued, in addition to those that may be caused by a regular memory store. The **Exceptions/Errors from Unsuccessful Memory Store (Exclusive) Operations** table lists these exceptions and errors.

Table 8-42: Exceptions/Errors from Unsuccessful Memory Store (Exclusive) Operations

Condition	Exception or Hardware Error
Access to misaligned address	The data access generated a misaligned address violation exception. The address for the exclusive access must be aligned. This restriction holds even if misaligned accesses are supported generally.
Access to core MMR	The data access attempted an illegal use of a supervisor resource.
Access to I/O device space	The data access generated a CPLB protection violation exception. This exception occurs when the access is to memory marked as I/O device space in the CPLB.
Access during in progress exclusive operation	The data access (exclusive write on memory bus) occurred while the SEQSTAT.XWACTIVE bit =1 or the SEQSTAT.XWAVAIL bit =1 before the new memory store (exclusive) instruction started.

16-Bit Store to Memory (StDregHToX16bit)

General Form

LdStExcl
cc = (w[PREG] = DREG_H) (excl)

Abstract

This instruction stores the most significant 16-bit value from a register to memory, using an exclusive memory write.

See Also (StDregToX08bit, StDregToX32bit, StDregLToX16bit)

StDregHToX16bit Description

The store high half data register to memory (16-bit transfer) checks for exclusive access to a memory location and writes a data value (contents of half register) to the location only if exclusive access is still held.

For more information about memory store (exclusive) operations, see Memory Store (Exclusive) Operations.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	**CC**	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

StDregHToX16bit Example

```
CC = (W[P4] = R3.H)(EXCL);  /* store exclusive from high 16-bits of a D-register */
```

16-Bit Store to Memory (StDregLToX16bit)

General Form

LdStExcl
cc = (w[PREG] = DREG) (excl)

Abstract

This instruction stores the least significant 16-bit value from a register to memory, using an exclusive memory write.

See Also (StDregToX08bit, StDregToX32bit, StDregHToX16bit)

StDregLToX16bit Description

The store low half data register to memory (16-bit transfer) checks for exclusive access to a memory location and writes a data value (contents of half register) to the location only if exclusive access is still held.

For more information about memory store (exclusive) operations, see Memory Store (Exclusive) Operations.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	**CC**	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

StDregLToX16bit Example

```
CC = (W[P4] = R3)(EXCL);  /* store exclusive from low 16-bits of a D-register */
CC = (W[P4] = R3.L)(EXCL);  /* alternate syntax for same instruction */
```

8-Bit Store to Memory (StDregToX08bit)

General Form

LdStExcl
cc = (b[PREG] = DREG) (excl)

Abstract

This instruction stores the least significant 8-bit value from a register to memory, using an exclusive memory write.

See Also (StDregToX32bit, StDregLToX16bit, StDregHToX16bit)

StDregToX08bit Description

The store data register to memory (8-bit transfer) checks for exclusive access to a memory location and writes a data value (least significant byte of data register contents) to the location only if exclusive access is still held.

For more information about memory store (exclusive) operations, see Memory Store (Exclusive) Operations.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	**CC**	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

StDregToX08bit Example

```
CC = (B[P4] = R6)(EXCL);  /* store exclusive from low 8-bits of a D-register */
```

32-Bit Store to Memory (StDregToX32bit)

General Form

LdStExcl
cc = ([PREG] = DREG) (excl)

Abstract

This instruction stores the 32-bit value from a register to memory, using an exclusive memory write.

See Also (StDregToX08bit, StDregLToX16bit, StDregHToX16bit)

StDregToX32bit Description

The store data register to memory (32-bit transfer) checks for exclusive access to a memory location and writes a data value (contents of register) to the location only if exclusive access is still held.

For more information about memory store (exclusive) operations, see Memory Store (Exclusive) Operations.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

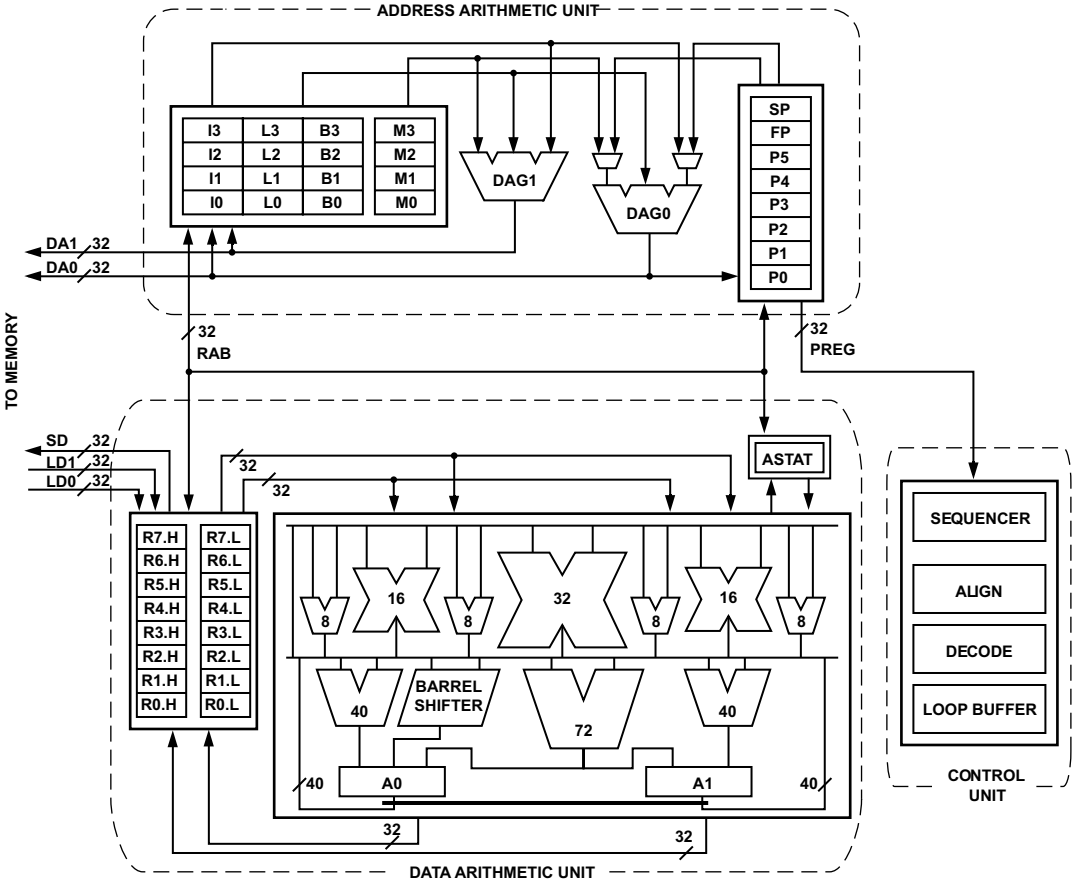
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	**CC**	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

StDregToX32bit Example

```
CC = ([P2] = R0)(EXCL);  /* store exclusive all 32-bits of a D-register */
```

Specialized Compute Instructions

The specialized compute instructions provide operations, which execute on the **data arithmetic unit** in the processor core. Users can take advantage of these instructions to detect exponents, add/subtract with prescale, divide, execute bitwise XOR, execute vector operations, execute Viterbi operations, and execute video related operations. These instructions are considered specialized because (unlike the arithmetic operation instructions) these instructions tend to provide features that are uniquely required to optimize specialized applications.



The operation types of specialized compute instructions include:

- Block Floating Point Operations
- DCT Operations
- Divide Operations
- Linear Feedback Shift Register LFSR Operations
- Video Operations
- Viterbi Operations

Block Floating Point Operations

These operations provide exponent adjustment for floating-point operations on register operands:

- Exponent Detection (Shift_ExpAdj32)

Exponent Detection (Shift_ExpAdj32)

General Form

Dsp32Shf
DREG_L = expadj (DREG , DREG_L)
DREG_L = expadj (DREG , DREG_L) (v)
DREG_L = expadj (DREG_L , DREG_L)
DREG_L = expadj (DREG_H , DREG_L)

Abstract

This instruction identifies the largest magnitude of a fractional number (YOP) and a reference exponent and returns the smaller of the two exponents. The exponent is the number of sign bits minus one. Exponents are unsigned integers. The input values can be a 32-bit register, a 16-bit half register, or a 16-bit vector.

Shift_ExpAdj32 Description

The exponent detection instruction identifies the largest magnitude of two or three fractional numbers based on their exponents. It compares the magnitude of one or two sample values to a reference exponent and returns the smallest of the exponents.

The exponent is the number of sign bits minus one. In other words, the exponent is the number of redundant sign bits in a signed number. Exponents are unsigned integers. The exponent detection instruction accommodates the two special cases (0 and –1) and always returns the smallest exponent for each case.

The reference exponent and destination exponent are 16-bit half-word unsigned values. The sample number can be either a word or half-word. The exponent detection instruction does not implicitly modify input values. The *dest_reg* and *exponent_register* can be the same data register. Doing this explicitly modifies the *exponent_register*.

The valid range of exponents is 0 through 31, with 31 representing the smallest 32-bit number magnitude and 15 representing the smallest 16-bit number magnitude.

Exponent detection supports three types of samples—one 32-bit sample, one 16-bit sample (either upper-half or lower-half word), and two 16-bit samples that occupy the upper-half and lower-half words of a single 32-bit register.

One special application of EXPADJ is to use this instruction to detect the exponent of the largest magnitude number in an array. The detected value may then be used to normalize the array on a subsequent pass with a shift operation. Typically, use this feature to implement block floating-point capabilities.

This **16-bit instruction** may be issued in parallel with certain other 16-bit other instructions.

This instruction may be used in either **User or Supervisor mode**.

Shift_ExpAdj32 Example

```
r5.l = expadj (r4, r2.l) ;
/* ...  Assume R4 = 0x0000 0052 and R2.L = 12. Then R5.L becomes 12. */
/* ...  Assume R4 = 0xFFFF 0052 and R2.L = 12. Then R5.L becomes 12. */
/* ...  Assume R4 = 0x0000 0052 and R2.L = 27. Then R5.L becomes 24. */
/* ...  Assume R4 = 0xF000 0052 and R2.L = 27. Then R5.L becomes 3. */


r5.l = expadj (r4.l, r2.l) ;
/* ...  Assume R4.L = 0x0765 and R2.L = 12. Then R5.L becomes 4. */
/* ...  Assume R4.L = 0xC765 and R2.L = 12. Then R5.L becomes 1. */


r5.l = expadj (r4.h, r2.l) ;
/* ...  Assume R4.H = 0x0765 and R2.L = 12. Then R5.L becomes 4. */
/* ...  Assume R4.H = 0xC765 and R2.L = 12. Then R5.L becomes 1. */


r5.l = expadj (r4, r2.l)(v) ;
/* ...  Assume R4.L = 0x0765, R4.H = 0xFF74 and R2.L = 12. Then R5.L becomes 4. */
/* ...  Assume R4.L = 0x0765, R4.H = 0xE722 and R2.L = 12. Then R5.L becomes 2. */
```

DCT Operations

These operations provide addition and/or subtract operations with prescale and rounding on register operands:

- 32-Bit Prescale Up Add/Sub to 16-bit (AddSubRnd12)
- 32-Bit Prescale Down Add/Sub to 16-Bit (AddSubRnd20)

32-Bit Prescale Up Add/Sub to 16-bit (AddSubRnd12)

General Form

Dsp32Alu
DDST0_HL = DREG + DREG (rnd12)
DDST0_HL = DREG - DREG (rnd12)

Abstract

This instruction shifts then adds or subtracts two 32-bit numbers, then it extracts sixteen bits of result. The instruction pre-shifts the operands four bits to the left, then it does the add/sub, round, and extract. The instruction supports only biased rounding, which adds a half LSB (bit 15) before truncating bits 15-0. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

See Also (AddSubRnd20)

AddSubRnd12 Description

The add/subtract prescale up instruction combines two 32-bit values to produce a 16-bit result as follows:

- Prescale up both input operand values by shifting them four places to the left
- Add or subtract the operands, depending on the instruction version used
- Round and saturate the upper 16 bits of the result
- Extract the upper 16 bits to the *dest_reg*

The instruction supports only biased rounding. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction. See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

A special applications of the add/subtract prescale up instruction is to use this instruction to provide an IEEE 1180–compliant 2D 8x8 inverse discrete cosine transform.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	**VS**	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubRnd12 Example

```
r1.l = r6+r7(rnd12) ;
r1.l = r6-r7(rnd12) ;
r1.h = r6+r7(rnd12) ;
r1.h = r6-r7(rnd12) ;
```

32-Bit Prescale Down Add/Sub to 16-Bit (AddSubRnd20)

General Form

Dsp32Alu
DDST0_HL = DREG + DREG (rnd20)
DDST0_HL = DREG - DREG (rnd20)

Abstract

This instruction shifts then adds or subtracts two 32-bit numbers, then it extracts sixteen bits of result. The instruction arithmetically pre-shifts the operands four bits to the right. It adds or subtracts them, rounds the upper 16-Bits of the result then extracts the upper 16-Bits to the result. The instruction supports only biased rounding, which adds a half LSB (bit 15) before truncating bits 15-0. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

See Also (AddSubRnd12)

AddSubRnd20 Description

The add/subtract prescale down instruction combines two 32-bit values to produce a 16-bit result as follows:

- Prescale down both input operand values by arithmetically shifting them four places to the right
- Add or subtract the operands, depending on the instruction version used
- Round the upper 16 bits of the result
- Extract the upper 16 bits to the *dest_reg*

The instruction supports only biased rounding. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction. See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This **32-bit instruction** can be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

A special applications of the add/subtract prescale down instruction is to use this instruction to provide an IEEE 1180–compliant 2D 8x8 inverse discrete cosine transform.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	**V**	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	**AN**	**AZ**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubRnd20 Example

```
r1.l = r6+r7(rnd20) ;
r1.l = r6-r7(rnd20) ;
r1.h = r6+r7(rnd20) ;
r1.h = r6-r7(rnd20) ;
```

Divide Operations

These operations provide division primitive operations on register operands:

- DIVS and DIVQ Divide Primitives (Divide)

DIVS and DIVQ Divide Primitives (Divide)

General Form

ALU2op
divq (DREG , DREG)
divs (DREG , DREG)

Abstract

The DIVQ instruction is a simple non-restoring divide primitive. It takes two operands, the first operand register is the dividend (numerator) and the second operand register is the divisor (denominator). The denominator or divisor is subtracted or added repeatedly from the numerator which becomes the dividend. The algorithm uses a status bit AQ (quotient bit), which determines how the ALU will compute the next bit of the quotient. If the AQ bit is 1 then an add is performed otherwise the dividend is subtracted from the partial remainder. The DIVS instruction is the initializing instruction for DIVQ. It sets the AQ flag based on the signs of the 32-bit dividend and the 16-bit divisor, left shifts the dividend one bit, then copies AQ into the dividend LSB.

Divide Description

The Divide Primitive instruction versions are the foundation elements of a nonrestoring conditional add-subtract division algorithm. See “Example” on page 15-24 for such a routine.

The dividend (numerator) is a 32-bit value. The divisor (denominator) is a 16-bit value in the lower half of divisor_register. The high-order half-word of divisor_register is ignored entirely.

The division can either be signed or unsigned, but the dividend and divisor must both be of the same type. The divisor cannot be negative. A signed division operation, where the dividend may be negative, begins the sequence with the DIVS (“divide-sign”) instruction, followed by repeated execution of the DIVQ (“divide-quotient”) instruction. An unsigned division omits the DIVS instruction. In that case, the user must manually clear the AQ status bit of the ASTAT register before issuing the DIVQ instructions.

Up to 16 bits of signed quotient resolution can be calculated by issuing DIVS once, then repeating the DIVQ instruction 15 times. A 16-bit unsigned quotient is calculated by omitting DIVS, clearing the AQ status bit, then issuing 16 DIVQ instructions.

Less quotient resolution is produced by executing fewer DIVQ iterations.

The result of each successive addition or subtraction appears in dividend_register, aligned and ready for the next addition or subtraction step. The contents of divisor_register are not modified by this instruction.

The final quotient appears in the low-order half-word of dividend_register at the end of the successive add/subtract sequence.

DIVS computes the sign bit of the quotient based on the signs of the dividend and divisor. DIVS initializes the AQ status bit based on that sign, and initializes the dividend for the first addition or subtraction. DIVS performs no addition or subtraction.

DIVQ either adds (dividend + divisor) or subtracts (dividend – divisor) based on the AQ status bit, then reinitializes the AQ status bit and dividend for the next iteration. If AQ is 1, addition is performed; if AQ is 0, subtraction is performed.

See "Status Bits Affected" on page 15-4 for the conditions that set and clear the AQ status bit.

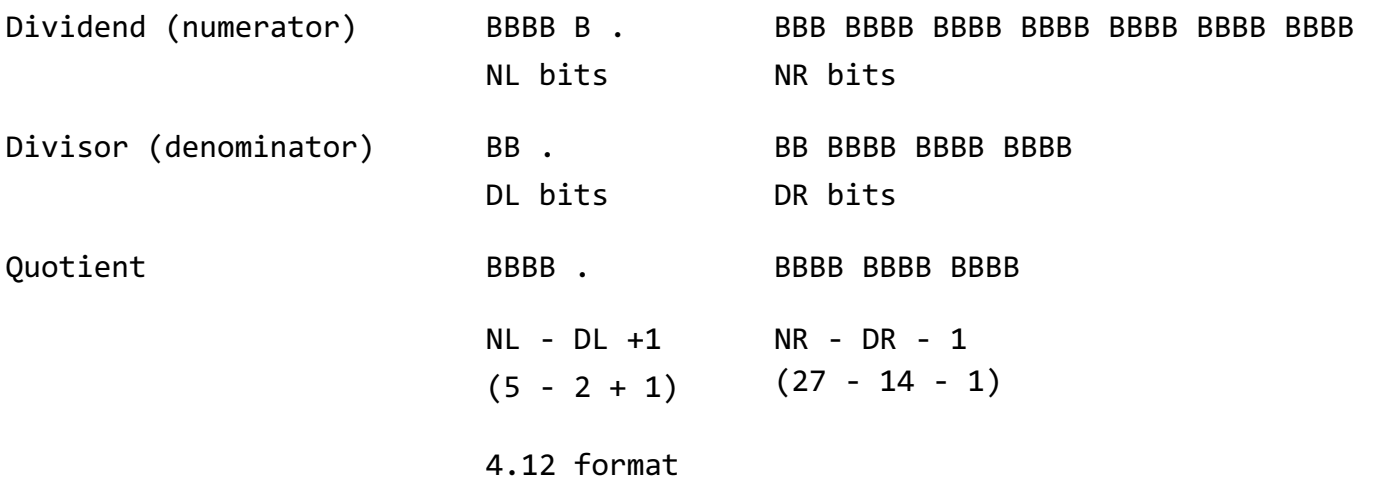
Both instruction versions align the dividend for the next iteration by left shifting the dividend one bit to the left (without carry). This left shift accomplishes the same function as aligning the divisor one bit to the right, such as one would do in manual binary division.

The format of the quotient for any numeric representation can be determined by the format of the dividend and divisor. Let:

- NL represent the number of bits to the left of the binal point of the dividend, and
- NR represent the number of bits to the right of the binal point of the dividend (numerator);
- DL represent the number of bits to the left of the binal point of the divisor, and
- DR represent the number of bits to the right of the binal point of the divisor (denominator).

Then the quotient has NL – DL + 1 bits to the left of the binal point and NR – DR – 1 bits to the right of the binal point. See the following example.

Figure 8-29: 4.12 Format



Some format manipulation may be necessary to guarantee the validity of the quotient. For example, if both operands are signed and fully fractional (dividend in 1.31 format and divisor in 1.15 format), the result is

fully fractional (in 1.15 format) and therefore the upper 16 bits of the dividend must have a smaller magnitude than the divisor to avoid a quotient overflow beyond 16 bits. If an overflow occurs, AV0 is set. User software is able to detect the overflow, rescale the operand, and repeat the division.

Dividing two integers (32.0 dividend by a 16.0 divisor) results in an invalid quotient format because the result will not fit in a 16-bit register. To divide two integers (dividend in 32.0 format and divisor in 16.0 format) and produce an integer quotient (in 16.0 format), one must shift the dividend one bit to the left (into 31.1 format) before dividing. This requirement to shift left limits the usable dividend range to 31 bits. Violations of this range produce an invalid result of the division operation.

The algorithm overflows if the result cannot be represented in the format of the quotient as calculated above, or when the divisor is zero or less than the upper 16 bits of the dividend in magnitude (which is tantamount to multiplication).

It is important to understand error conditions related to this instruction. Two special cases can produce invalid or inaccurate results. Software can trap and correct both cases.

No signed division by a negative divisor

The Divide Primitive instructions do not support signed division by a negative divisor. Attempts to divide by a negative divisor result in a quotient that is, in most cases, one LSB less than the correct value. If division by a negative divisor is required, follow the steps below.

a. Before performing the division, save the sign of the divisor in a scratch register.

b. Calculate the absolute value of the divisor and use that value as the divisor operand in the Divide Primitive instructions.

c. After the divide sequence concludes, multiply the resulting quotient by the original divisor sign.

d. The quotient then has the correct magnitude and sign.

No unsigned division by a divisor greater than 0x7FFF

The Divide Primitive instructions do not support unsigned division by a divisor greater than 0x7FFF. If such divisions are necessary, prescale both operands by shifting the dividend and divisor one bit to the right prior to division. The resulting quotient will be correctly aligned. Of course, prescaling the operands decreases their resolution, and may introduce one LSB of error in the quotient. Such error can be detected and corrected by the following steps.

a. Save the original (unscaled) dividend and divisor in scratch registers.

b. Prescale both operands as described and perform the division as usual.

c. Multiply the resulting quotient by the unscaled divisor. Do not corrupt the quotient by the multiplication step.

d. Subtract the product from the unscaled dividend. This step produces an error value.

e. Compare the error value to the unscaled divisor.

f. If error > divisor, add one LSB to the quotient.

g. If error < divisor, subtract one LSB from the quotient.

h. If error = divisor, do nothing.

This **16-bit instruction** may not be issued in parallel with other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Divide Example

```
/* Evaluate given a signed integer dividend and divisor */
p0 = 15 ; /* Evaluate the quotient to 16 bits. */
r0 = 70 ; /* Dividend, or numerator */
r1 = 5 ; /* Divisor, or denominator */
r0 <<= 1 ; /* Left shift dividend by 1 needed for integer division */
divs (r0, r1) ; /* Evaluate quotient MSB. Initialize AQ status bit and dividend for
the DIVQ loop. */
loop .div_prim lc0=p0 ; /* Evaluate DIVQ p0=15 times. */
loop_begin .div_prim ;
divq (r0, r1) ;
loop_end .div_prim ;
r0 = r0.l (x) ; /* Sign extend the 16-bit quotient to 32bits. */
/* r0 contains the quotient (70/5 = 14). */
```

Linear Feedback Shift Register LFSR Operations

These operations provide LFSR related operations on register operands:

- 40-Bit BXORShift LSFR with Feedback to the Accumulator (BXORShift_NF)
- 40-Bit BXOR LSFR with Feedback to a Register (BXOR)
- 32-Bit BXOR or BXORShift LSFR without Feedback (BXOR_NF)

40-Bit BXOR LSFR with Feedback to a Register (BXOR)

General Form

Dsp32Shf
DREG_L = cc = bxor (a0, a1, cc)

Abstract

This instruction (linear feedback shift register, LFSR) provides a bit-wise XOR reduction of A0 logically AND'ed with A1 and the feedback bit (CC). The result is placed into both the CC flag and the least significant bit of the destination register. The Accumulator is not modified by this operation.

See Also (BXORShift_NF, BXOR_NF)

BXOR Description

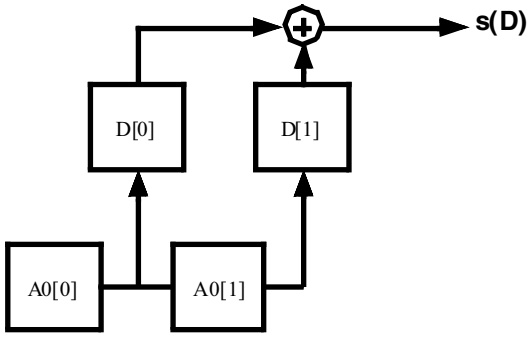
Four Bit-Wise Exclusive-OR (BXOR) instructions support two different types of linear feedback shift register (LFSR) implementations. The Type I LFSRs (no feedback) applies a 32-bit registered mask to a 40-bit state residing in Accumulator A0, followed by a bit-wise XOR reduction operation. The result is placed in CC and a destination register half. The Type I LFSRs (with feedback) applies a 40-bit mask in Accumulator A1 to a 40-bit state residing in A0. The result is shifted into A0. In the following circuits describing the BXOR instruction group, a bit-wise XOR reduction is defined as:

Figure 8-30:

$$Out = (((((B_0 \oplus B_1) \oplus B_2) \oplus B_3) \oplus ...) \oplus B_{n-1})$$

where B0 through BN–1 represent the N bits that result from masking the contents of Accumulator A0 with the polynomial stored in either A1 or a 32-bit register. The instruction descriptions are shown in Figure 12-1.

Figure 8-31: Bit-Wise Exclusive-OR Reduction



In the figure above, the bits A0 bit 0 and A0 bit 1 are logically AND'ed with bits D[0] and D[1]. The result from this operation is XOR reduced according to the following formula.

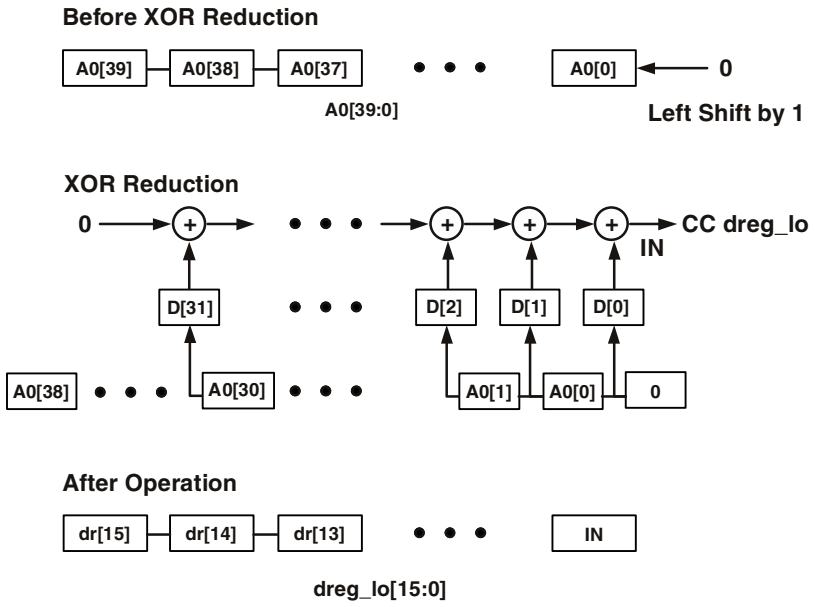
Figure 8-32:

$$s(D) = (A0[0]\&D[0]) \oplus (A0[1]\&D[1])$$

Modified Type I LFSR (without feedback) Two instructions support the LSFR with no feedback. Dreg_lo = CC = BXORSHIFT(A0, dreg) Dreg_lo = CC = BXOR(A0, dreg) In the first instruction the Accumulator A0 is left-shifted by 1 prior to the XOR reduction. This instruction provides a bit-wise XOR of A0 logically AND'ed with a dreg. The result of the operation is placed into both the CC status bit and the least significant bit of the destination register. The operation is shown in Figure 12-2. The upper 15 bits of dreg_lo are overwritten with zero, and dr[0] = IN after the operation.

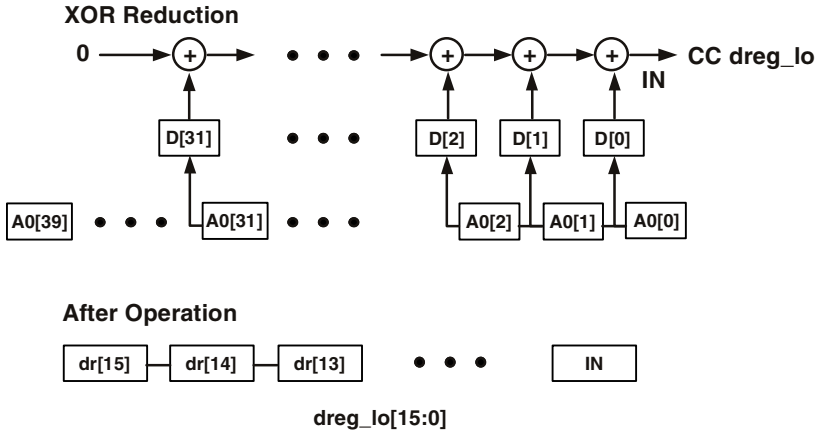
Figure 12-2.

Figure 8-33: A0 Left-Shifted by 1 Followed by XOR Reduction



The second instruction in this class performs a bit-wise XOR of A0 logically AND'ed with the dreg. The output is placed into the least significant bit of the destination register and into the CC bit. The Accumulator A0 is not modified by this operation. This operation is illustrated in Figure 12-3. The upper 15 bits of dreg_lo are overwritten with zero, and dr[0] = IN after the operation.

Figure 8-34: XOR of A0, Logical AND with the D-Register

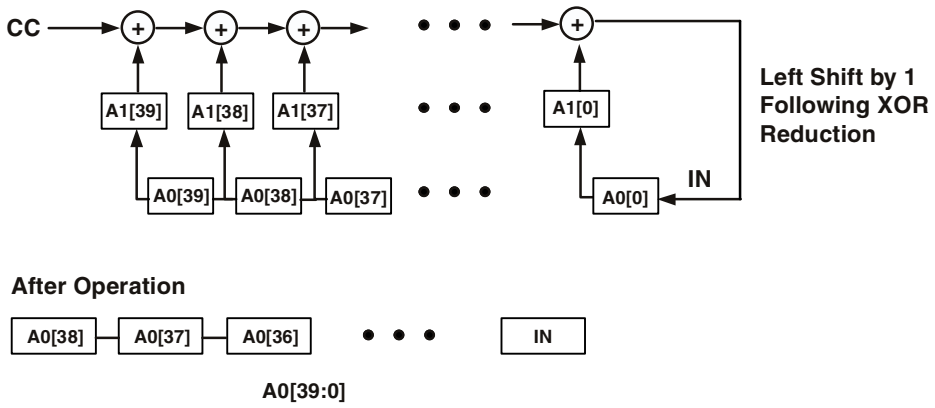


Modified Type I LFSR (with feedback) Two instructions support the LFSR with feedback. A0 = BXOR-SHIFT(A0, A1, CC) Dreg_lo = CC = BXOR(A0, A1, CC)

The first instruction provides a bit-wise XOR of A0 logically AND'ed with A1. The resulting intermediate bit is XOR'ed with the CC status bit. The result of the operation is left-shifted into the least significant bit

of A0 following the operation. This operation is illustrated in Figure 12-4. The CC bit is not modified by this operation.

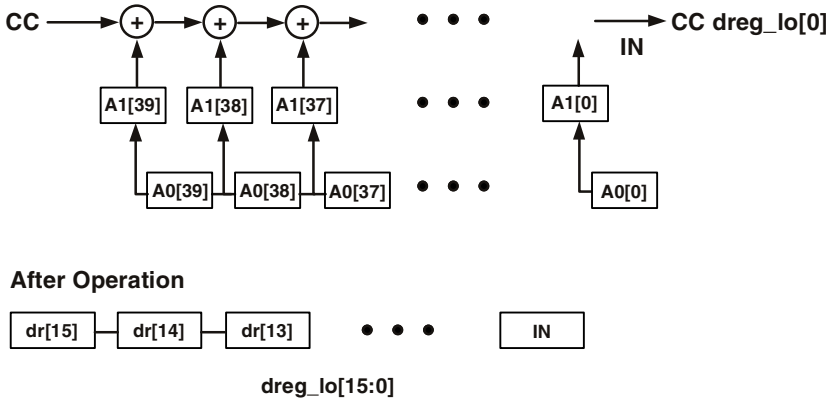
Figure 8-35: XOR of A0 AND A1, Left-Shifted into LSB of A0



The second instruction in this class performs a bit-wise XOR of A0 logically AND'ed with A1. The resulting intermediate bit is XOR'ed with the CC status bit. The result of the operation is placed into both the CC status bit and the least significant bit of the destination register.

This operation is illustrated in Figure 12-5.

Figure 8-36: XOR of A0 AND A1, to CC Bit and LSB of Dest Register



The Accumulator A0 is not modified by this operation. The upper 15 bits of dreg_lo are overwritten with zero, and dr[0] = IN.

Special Applications Linear feedback shift registers (LFSRs) can multiply and divide polynomials and are often used to implement cyclical encoders and decoders.

LFSRs use the set of Bit-Wise XOR instructions to compute bit XOR reduction from a state masked by a polynomial.

When implementing a CRC algorithm, it is known that there is an equivalence between polynomial division and LFSR circuits. For example, CRC is defined as the remainder of the division of a message polynomial appended with n zeros by the code generator polynomial:

$C_n(x)=\{M_k(x)x^n\}\text{mod}G_n(x)$

Where:

- $M_{k-1}(x)$ is the message polynomial of length k:

  $M_{k-1}(x)=m_{k-1}x^{k-1}+m_{k-2}x^{k-2}+\ldots+m_0x^0$

- $G_n(x)$ is the CRC generating polynomial of degree n, and n is also the CRC field length in bits:

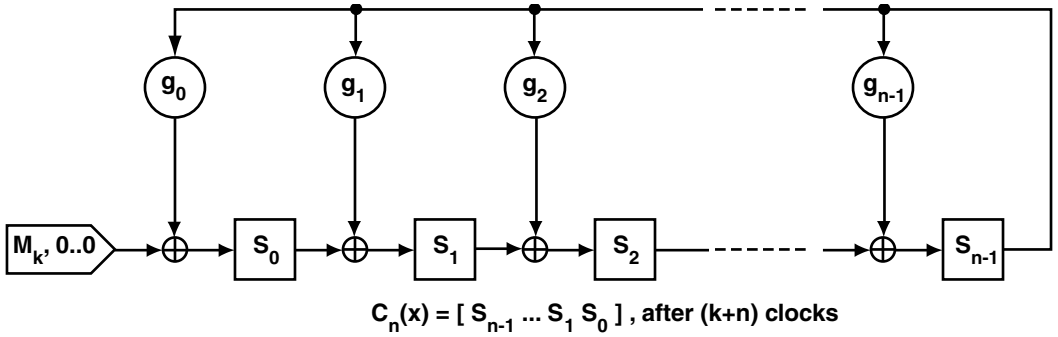
  $G_n(x)=x^n+g_{n-1}x^{n-1}+\ldots+g_0x^0$

- $C_n(x)$ is the calculated CRC polynomial of degree n:

  $C_n(x)=x^n+c_{n-1}x^{n-1}+\ldots+c_0x^0$

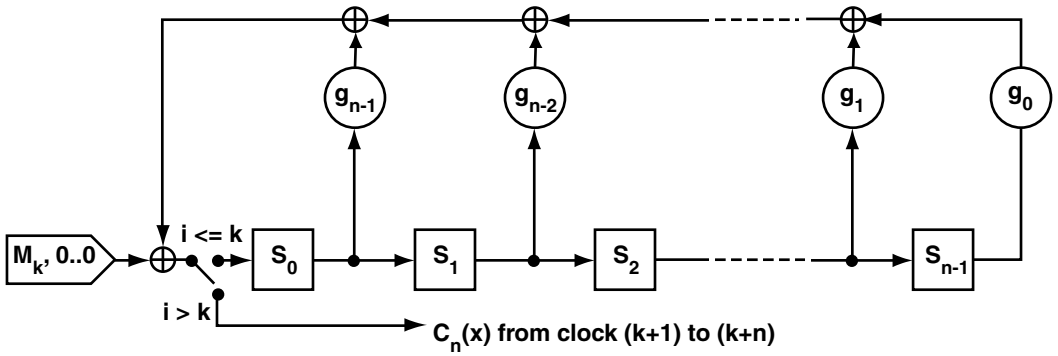
The division is performed modulo-2 over Galois field GF2. In the above equation, the message stream Mk is postfixed by n zeros before the actual division. This equation can be implemented by one of two types of n taps LFSR's. The more familiar type of LFSR is called Type II (or internal) LFSR of the form:

Figure 8-37: Internal LFSR (Type II)



The other type of LFSR, Type I (or external LFSR) has the form:

Figure 8-38: External LFSR (Type I)



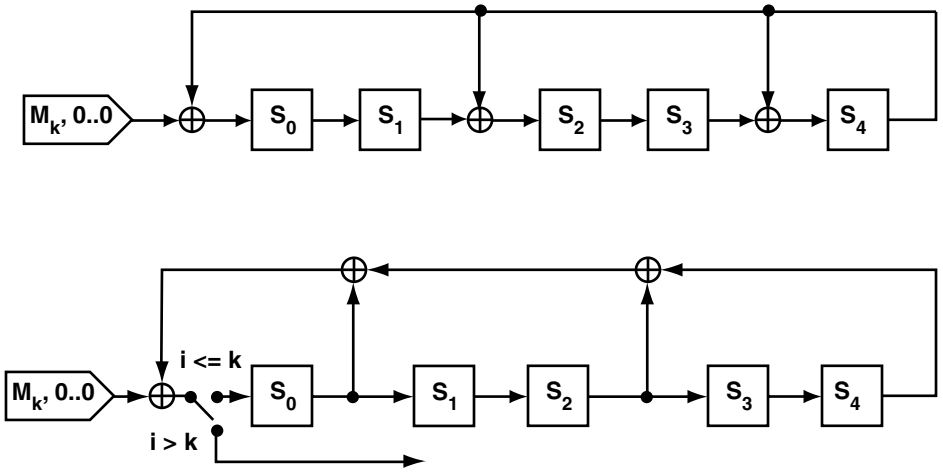
The two are equivalent, and the simple rule for conversion from Type II to Type I is:

1. While keeping the LFSR flow direction, flip the order of the feedback taps.

2. After the first k clocks, feed the first tap (S0) with n zeros and read the n output bits (which are the required CRC) as the sum of the feedback and the input.

For example, consider the following equivalent implementations of the polynomial G5(x) = x5+x4+x2+1:

Figure 8-39: Internal (Type II) Versus External (Type I) LFSR



This **16-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BXOR Example

The BXOR and BXORSHIFT instructions let you calculate a Type I CRC at a rate of two cycles per input bit, as in the following example program.

```
// _CRC_BXOR - calculate CRC value of a message polynomial
// for a given generator polynomial.
#define MSG_LEN 32 // bits
```

```
#define CRC_LEN 16 // bits
_CRC_BXOR:
   a1 = a0 = 0;
   r1 = 0x8408 (z); // LFSR polynomial, reversed:
   // x^16 + x^12 + x^5 + 1
   a1.w = r1; // initialize LFSR mask
   r2.h = 0xd065; // r2 = message
   r2.l = 0x86c9;
   p1 = MSG_LEN (z);
   loop _MSG_loop lc0 = p1;
   loop_begin _MSG_loop;
      r2 = rot r2 by 1;
      a0 = bxorshift(a0, a1, cc);
   loop_end _MSG_loop;
   r0 = 0; // initialize CRC
   r2.l = cc = bxor(a0, r1);
   r0 = rot r0 by 1;
   p1 = CRC_LEN-1 (z);
   loop _CRC_loop lc0 = p1;
   loop_begin _CRC_loop;
      r2.l = cc = bxorshift(a0, r1);
      r0 = rot r0 by 1;
   loop_end _CRC_loop;
   // r0.l now contains the CRC
_CRC_BXOR.end:
```

40-Bit BXORShift LSFR with Feedback to the Accumulator (BXORShift_NF)

General Form

Dsp32Shf
a0 = bxorshift (a0, a1, cc)

Abstract

This instruction (linear feedback shift register, LFSR) provides a bit-wise XOR reduction of A0 logically AND'ed with A1 and the feedback bit (CC). The result is left-shifted into the least significant bit of A0. The CC bit is not modified.

See Also (BXOR, BXOR_NF)

BXORShift_NF Description

Linear feedback shift register (LFSR) instruction. Provides a bit-wise XOR reduction of A0 logically AND'ed with A1 and the feedback bit (CC). The result is left-shifted into the least significant bit of A0. The CC bit is not modified.

The bit-wise XOR reduction is defined as:

$$out = (((((CC \oplus B_0) \oplus B_1) \oplus B_2) \oplus ...) \oplus B_{n-1})$$

For more information about this instruction, see 40-Bit BXOR LSFR with Feedback to a Register (BXOR).

BXORShift_NF Example

For examples using this instruction, see 40-Bit BXOR LSFR with Feedback to a Register (BXOR).

32-Bit BXOR or BXORShift LSFR without Feedback (BXOR_NF)

General Form

Dsp32Shf
DREG_L = cc = bxorshift (a0, DREG)
DREG_L = cc = bxor (a0, DREG)

Abstract

This instruction (linear feedback shift register, LFSR) provides a bit-wise XOR reduction of A0 or A0 shifted left one, logically AND'ed with a 32-bit data register. The result is placed into both the CC flag and the least significant bit of the destination register.

See Also (BXORShift_NF, BXOR)

BXOR_NF Description

Linear feedback shift register (LFSR) instruction. Provides a bit-wise XOR reduction of A0 or A0 shifted left one, logically AND'ed with a 32-bit data register. The result is placed into both the CC flag and the least significant bit of the destination register.

A bit-wise XOR reduction is defined as:

$out = \%\%(((((\%\%B_0 \ \&oplus\ B_1) \ \&oplus\ B_2) \ \&oplus\ B_3) \ \&oplus\ ...) \ \&oplus\ B_{n-1})$

For more information about this instruction, see 40-Bit BXOR LSFR with Feedback to a Register (BXOR).

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	**CC**	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BXOR_NF Example

For examples using this instruction, see 40-Bit BXOR LSFR with Feedback to a Register (BXOR).

Video Operations

These operations provide video application specific operations on register operands:

- Vectored 8-Bit Add or Subtract to 16-Bit (Byteop16P/M) (AddSub4x8)
- Vectored 8-Bit to 16-Bit Add then Clip to 8-Bit (Byteop3P) (AddClip)
- Disable Alignment Exception (DisAlignExcept)
- Quad Byte Average (Byteop2P) (Avg4x8Vec)
- Vector Byte Average (Byteop1P) (Avg8Vec)
- Dual Accumulator Extraction with Addition (AddAccHalf)
- Vectored 8-Bit Sum of Absolute Differences (SAD8Vec)

Vectored 8-Bit to 16-Bit Add then Clip to 8-Bit (Byteop3P) (AddClip)

General Form

Dsp32Alu
DREG = byteop3p (PAIR0 , PAIR1) (lo RSC)
DREG = byteop3p (PAIR0 , PAIR1) (hi RSC)

Abstract

This instruction adds two 8-bit unsigned values to two 16-bit signed values, then it clips the result back to the 8-bit unsigned range. The instruction either adds Y[3] & Y[1] or Y[2] & Y[0] to the two 16-bit X values. The lower two bits of I0 and I1 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair. The results are written back to either the lower or higher bytes of each 16-bit result half. The unused bytes are filled with zeros.

See Also (AddSub4x8)

AddClip Description

The dual 16-bit add/clip instruction adds two 8-bit unsigned values to two 16-bit signed values, then limits (or “clips”) the result to the 8-bit unsigned range 0 through 255, inclusive. The instruction loads the results

as bytes on half-word boundaries in one 32-bit destination register. Some syntax options load the upper byte in the half-word and others load the lower byte, as shown in the next few figures.

Figure 8-40: The source registers contain:

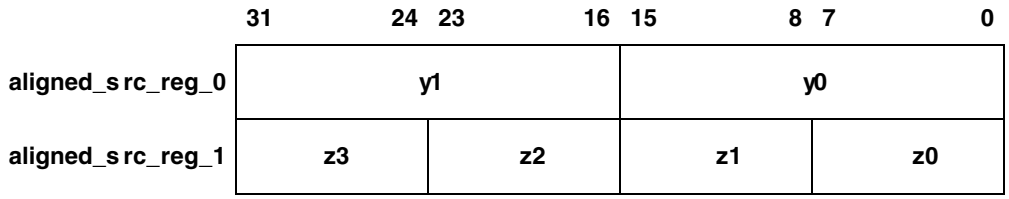


Figure 8-41: The versions that load the result into the lower byte (LO) produce:

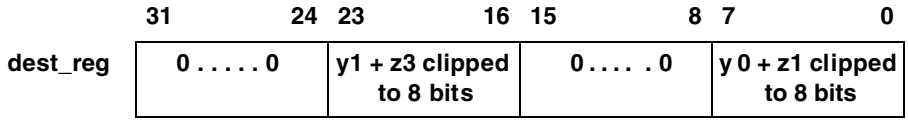
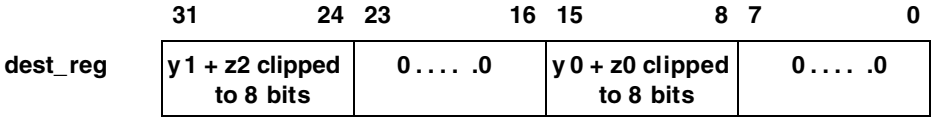


Figure 8-42: The versions that load the result into the higher byte (HI) produce:



In either case, the unused bytes in the destination register are filled with 0x00. The 8-bit and 16-bit addition is performed as a signed operation. The 16-bit operand is sign-extended to 32 bits before adding.

The only valid input source register pairs are R1:0 and R3:2.

The dual 16-bit add/clip instruction provides byte alignment directly in the source register pairs src_reg_0 and src_reg_1 based on index registers I0 and I1.

- The two LSBs of the I0 register determine the byte alignment for source register pair src_reg_0 (typically R1:0).
- The two LSBs of the I1 register determine the byte alignment for source register pair src_reg_1 (typically R3:2).

The relationship between the I-register bits and the byte alignment is illustrated in the **I-register Bits and the Byte Alignment (no reverse)** figure.

In the default source order case (for example, not the (– , R) syntax), assuming a source register pair contains the following.

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

Figure 8-43: I-register Bits and the Byte Alignment (no reverse)

The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

Options The (– , R) syntax reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (– , R) option causes the low order bytes to come from the high register.

In the optional reverse source order case (for example, using the (– , R) syntax), the only difference is the source registers swap places within the register pair in their byte ordering. Assume a source register pair contains the data shown in the **I-register Bits and the Byte Alignment (with reverse)** figure.

Figure 8-44: I-register Bits and the Byte Alignment (with reverse)

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

A special application of this instruction is support for video motion compensation algorithms. The instruction supports the addition of the residual to a video pixel value, followed by unsigned byte saturation.

This **16-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

AddClip Example

```
r3 = byteop3p (r1:0, r3:2) (lo) ;
r3 = byteop3p (r1:0, r3:2) (hi) ;
```

```
r3 = byteop3p (r1:0, r3:2) (lo, r) ;
r3 = byteop3p (r1:0, r3:2) (hi, r) ;
```

Vectored 8-Bit Add or Subtract to 16-Bit (Byteop16P/M) (AddSub4x8)

General Form

Dsp32Alu
(DREG , DREG) = byteop16p (PAIR0 , PAIR1) RS
(DREG , DREG) = byteop16m (PAIR0 , PAIR1) RS

Abstract

This instruction (Byteop16M and ByteOp16P) adds or subtracts two unsigned quad byte vectors, adjusting for byte alignment. The lower two bits of I0 and I1 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair

See Also (AddClip)

AddSub4x8 Description

The quad 8-bit add instruction adds two unsigned quad byte number sets byte-wise, adjusting for byte alignment. It then loads the byte-wise results as 16-bit, zero-extended, half-words in two destination registers, as shown in the next figures.

Figure 8-45: Source Registers Contain

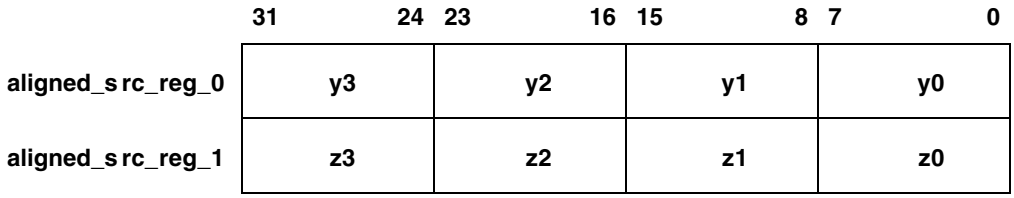
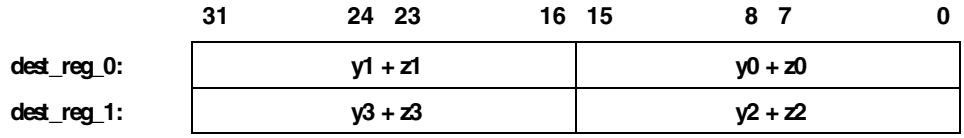


Figure 8-46: Destination Registers Receive



The only valid input source register pairs are R1:0 and R3:2, and the two destination registers must be unique.

The Quad 8-Bit Add instruction provides byte alignment directly in the source register pairs src_reg_0 and src_reg_1 based on index registers I0 and I1.

- The two LSBs of the I0 register determine the byte alignment for source register pair src_reg_0 (typically R1:0).
- The two LSBs of the I1 register determine the byte alignment for source register pair src_reg_1 (typically R3:2).

The relationship between the I-register bits and the byte alignment is illustrated in Table 18-1.

In the default source order case (for example, not the (R) syntax), assume that a source register pair contains the data shown in the **I-register Bits and the Byte Alignment (No Reverse)** figure.

Figure 8-47: I-register Bits and the Byte Alignment (No Reverse)

The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

Options

The (R) syntax reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.

In the optional reverse source order case (for example, using the (R) syntax), the only difference is the source registers swap places within the register pair in their byte ordering. Assume a source register pair contains the data shown in the **I-register Bits and the Byte Alignment (With Reverse)** figure.

Figure 8-48: I-register Bits and the Byte Alignment (With Reverse)

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

The mnemonic derives its name from the fact that the operands are bytes, the result is 16 bits, and the arithmetic operation is "plus" for addition.

A special application of this instruction provides packed data arithmetic typical of video and image processing applications.

This **16-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

AddSub4x8 Example

```
(r1,r2)= byteop16p (r3:2,r1:0) ;
(r1,r2)= byteop16p (r3:2,r1:0) (r) ;
```

Disable Alignment Exception (DisAlignExcept)

General Form

Dsp32Alu
disalgnexcpt

Abstract

This instruction disables alignment exceptions. This instruction only affects misaligned loads that use I registers. The address is forced to be 32-bit aligned.

See Also (Shift_Align)

DisAlignExcept Description

The disable alignment exception for load (DISALGNEXCPT) instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel. This instruction only affects misaligned 32-bit load instructions that use I-register indirect addressing.

In order to force address alignment to a 32-bit boundary, the two LSBs of the address are cleared before being sent to the memory system. The I-register is not modified by the DISALIGNEXCPT instruction. Also, any modifications performed to the I-register by a parallel instruction are not affected by the DISALIGNEXCPT instruction.

A special applications of this instruction is to use the DISALGNEXCPT instruction when priming data registers for Quad 8-Bit single-instruction, multiple-data (SIMD) instructions.

Quad 8-Bit SIMD instructions require as many as sixteen 8-bit operands, four D-registers worth, to be preloaded with operand data. The operand data is 8 bits and not necessarily word aligned in memory. Thus, use DISALGNEXCPT to prevent spurious exceptions for these potentially misaligned accesses.

During execution, when Quad 8-Bit SIMD instructions perform 8-bit boundary accesses, they automatically prevent exceptions for misaligned accesses. No user intervention is required.

This **16-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

DisAlignExcept Example

```
disalgnexcpt || r1 = [i0++] || r3 = [i1++] ;
/* three instructions in parallel */
disalgnexcpt || [p0 ++ p1] = r5 || r3 = [i1++] ;
/* alignment exception is prevented only for the load */
```

```
disalgnexcpt || r0 = [p2++] || r3 = [i1++] ;
/* alignment exception is prevented only for the I-reg load */
```

Byte Align (Shift_Align)

General Form

Dsp32Shf
DREG = align8 (DREG , DREG)
DREG = align16 (DREG , DREG)
DREG = align24 (DREG , DREG)

Abstract

This instruction copies four contiguous bytes from a register pair. The bytes are offset by 8, 16, or 24 bits in the register pair.

See Also (DisAlignExcept)

Shift_Align Description

The Byte Align instruction copies a contiguous four-byte unaligned word from a combination of two data registers. The instruction version determines the bytes that are copied; in other words, the byte alignment of the copied word. Alignment options are shown in Table 18-1.

Figure 8-49: Byte Alignment Options

	src_reg_1				src_reg_0			
	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
dest_reg for ALIGN8:				byte 4	byte 3	byte 2	byte 1	
dest_reg for ALIGN16:			byte 5	byte 4	byte 3	byte 2		
dest_reg for ALIGN24:		byte 6	byte 5	byte 4	byte 3			

The ALIGN16 version performs the same operation as the Vector Pack instruction using the syntax:

```
dest_reg = PACK ( Dreg_lo, Dreg_hi )
```

Use the Byte Align instruction to align data bytes for subsequent single- instruction, multiple-data (SIMD) instructions.

The input values are not implicitly modified by this instruction. The destination register can be the same D-register as one of the source registers. Doing this explicitly modifies that source register.

This **16-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

Shift_Align Example

```
// If r3 = 0x0011 2233 and r4 = 0x4455 6677, then . . .
r0 = align8 (r3, r4) ; /* produces r0 = 0x3344 5566, */
r0 = align16 (r3, r4) ; /* produces r0 = 0x2233 4455, and */
r0 = align24 (r3, r4) ; /* produces r0 = 0x1122 3344, */
```

Quad Byte Average (Byteop2P) (Avg4x8Vec)

General Form

Dsp32Alu
DREG = byteop2p (PAIR0 , PAIR1) (rndl RSC)
DREG = byteop2p (PAIR0 , PAIR1) (rndh RSC)
DREG = byteop2p (PAIR0 , PAIR1) (tl RSC)
DREG = byteop2p (PAIR0 , PAIR1) (th RSC)

Abstract

This instruction averages the upper two bytes and the lower two bytes of two unsigned quad byte vectors adjusting for byte alignment. The lower two bits of I0 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair. Note that this operation only uses I0, which is different than all the other byteop instructions. If you specify round (RND), a round bit is added prior to the shift. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction. It returns the two averages in either Dest[3] & Dest[1] or Dest[2] & Dest[0].

See Also (Avg8Vec)

Avg4x8Vec Description

The quad 8-bit average half-word instruction finds the arithmetic average of two unsigned quad byte number sets byte wise, adjusting for byte alignment. This instruction averages four bytes together. The instruction loads the results as bytes on half-word boundaries in one 32-bit destination register. Some

syntax options load the upper byte in the half-word and others load the lower byte, as shown in the next few figures.

Figure 8-50: Source Registers Contain

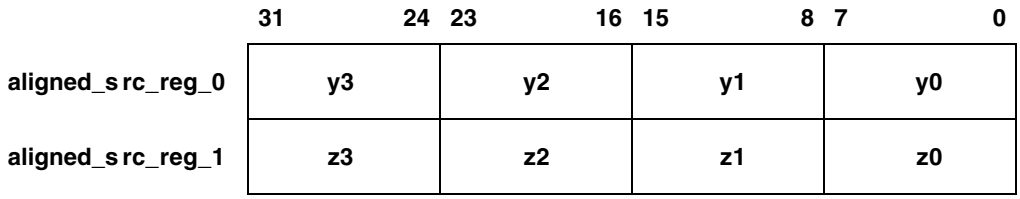


Figure 8-51: The versions that load the result into the lower byte – RNDL and TL – produce:

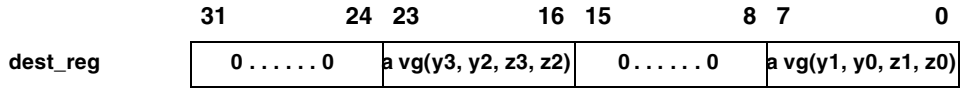
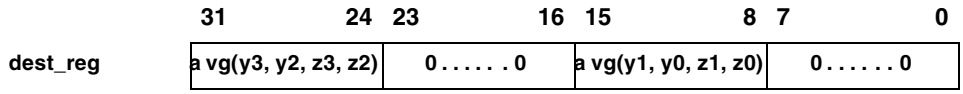


Figure 8-52: The versions that load the result into the higher byte – RNDH and TH – produce:



In either case, the unused bytes in the destination register are filled with 0x00.

Arithmetic average (or mean) is calculated by summing the four byte operands, then shifting right two places to divide by four.

When the intermediate sum is not evenly divisible by 4, precision may be lost.

The user has two options to bias the result–truncation or biased rounding.

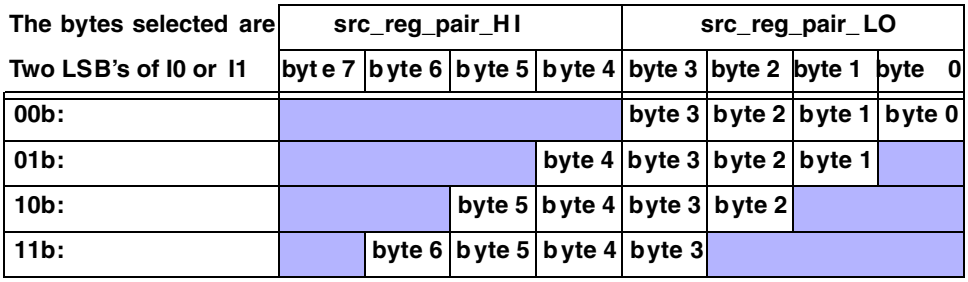
The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

The only valid input source register pairs are R1:0 and R3:2.

The quad 8-bit average half-word instruction provides byte alignment directly in the source register pairs src_reg_0 (typically R1:0) and src_reg_1 (typically R3:2) based only on the I0 register. The byte alignment in both source registers must be identical since only one register specifies the byte alignment for them both.

The relationship between the I-register bits and the byte alignment is shown in the **I-register Bits and the Byte Alignment (no reverse)** figure. In the default source order case (for example, not the (R) syntax), assume a source register pair contains the data shown in the figure.

Figure 8-53: I-register Bits and the Byte Alignment (no reverse)



This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

The quad 8-bit average half-word instruction supports the options shown in the **Options for Quad 8-Bit Average -- Half-Word** table.

Table 8-43: Options for Quad 8-Bit Average -- Half-Word

Option	Description
(RND—)	Rounds up the arithmetic mean.
(T—)	Truncates the arithmetic mean.
(—L)	Loads the results into the lower byte of each destination half-word.
(—H)	Loads the results into the higher byte of each destination half-word.
(—,R)	Reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.

When used together, the order of the options in the syntax makes no difference. In the optional reverse source order case (for example, using the (R) syntax), the only difference is the source registers swap places

within the register pair in their byte ordering. Assume a source register pair contains the data shown in the **I-register Bits and the Byte Alignment (with reverse)** figure.

Figure 8-54: I-register Bits and the Byte Alignment (with reverse)

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

The mnemonic derives its name from the fact that the operands are bytes, the result is two half-words, and the basic arithmetic operation is "plus" for addition. The single destination register indicates that averaging is performed.

A special applications of this instruction is support for binary interpolation used in fractional motion search and motion compensation algorithms.

This **16-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

Avg4x8Vec Example

```
r3 = byteop2p (r1:0, r3:2) (rndl) ;
r3 = byteop2p (r1:0, r3:2) (rndh) ;
r3 = byteop2p (r1:0, r3:2) (tl) ;
r3 = byteop2p (r1:0, r3:2) (th) ;
r3 = byteop2p (r1:0, r3:2) (rndl, r) ;
r3 = byteop2p (r1:0, r3:2) (rndh, r) ;
r3 = byteop2p (r1:0, r3:2) (tl, r) ;
r3 = byteop2p (r1:0, r3:2) (th, r) ;
```

Vector Byte Average (Byteop1P) (Avg8Vec)

General Form

Dsp32Alu
DREG = byteop1p (PAIR0 , PAIR1) RS
DREG = byteop1p (PAIR0 , PAIR1) (t RSC)

Abstract

This instruction computes the vector average of two unsigned quad byte vectors adjusting for byte alignment. The lower two bits of I0 and I1 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair. By default, this instruction rounds by adding a one prior to shifting. If you specify truncate, the round bit is not added. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

See Also (Avg4x8Vec)

Avg8Vec Description

The quad 8-bit average byte instruction computes the arithmetic average of two unsigned quad byte number sets byte wise, adjusting for byte alignment. This instruction loads the byte-wise results as concatenated bytes in one 32-bit destination register, as shown in the next figures.

Figure 8-55: Source Registers Contain

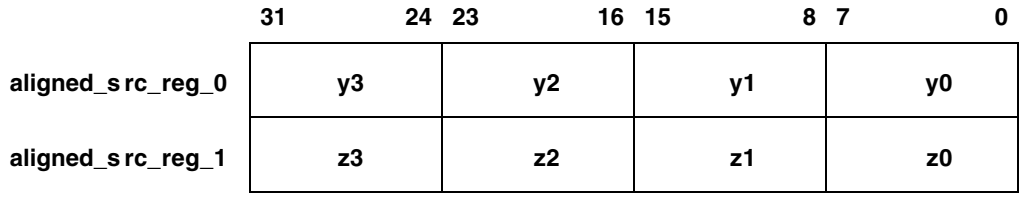
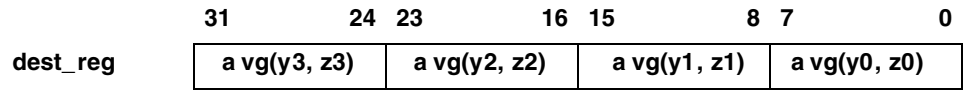


Figure 8-56: Destination Registers Receive



Arithmetic average (or mean) is calculated by summing the two operands, then shifting right one place to divide by two.

The user has two options to bias the result–truncation or rounding up. By default, the architecture rounds up the mean when the sum is odd. However, the syntax supports optional truncation.

See “Rounding and Truncating” on page 1-19 for a description of biased rounding and truncating behavior.

The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction. The only valid input source register pairs are R1:0 and R3:2.

The Quad 8-Bit Average – Byte instruction provides byte alignment directly in the source register pairs src_reg_0 and src_reg_1 based on index registers I0 and I1.

- The two LSBs of the I0 register determine the byte alignment for source register pair src_reg_0 (typically R1:0).
- The two LSBs of the I1 register determine the byte alignment for source register pair src_reg_1 (typically R3:2).

The relationship between the I-register bits and the byte alignment is illustrated in the **I-register Bits and the Byte Alignment (no reverse)** figure.

In the default source order case (for example, not the (R) syntax), assume a source register pair contains the data shown in the figure.

Figure 8-57: I-register Bits and the Byte Alignment (no reverse)

The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

Options

The quad 8-bit average byte instruction supports the options shown in the **Options for Quad 8-Bit Average – Byte** table.

Table 8-44: Options for Quad 8-Bit Average – Byte

Option	Description
Default	Rounds up the arithmetic mean.
(T)	Truncates the arithmetic mean.

Option	Description
(R)	Reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.
(T, R)	Combines both of the above options.

In the optional reverse source order case (for example, using the (R) syntax), the only difference is the source registers swap places within the register pair in their byte ordering. Assume a source register pair contains the data shown in the **I-register Bits and the Byte Alignment (with reverse)** figure.

Figure 8-58: I-register Bits and the Byte Alignment (with reverse)

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

The mnemonic derives its name from the fact that the operands are bytes, the result is one word, and the basic arithmetic operation is "plus" for addition. The single destination register indicates that averaging is performed.

A special application of this instruction is support for binary interpolation used in fractional motion search and motion compensation algorithms.

This **16-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

Avg8Vec Example

```
r3 = byteop1p (r1:0, r3:2) ;
r3 = byteop1p (r1:0, r3:2) (r) ;
r3 = byteop1p (r1:0, r3:2) (t) ;
r3 = byteop1p (r1:0, r3:2) (t,r) ;
```

Dual Accumulator Extraction with Addition (AddAccHalf)

General Form

Dsp32Alu
DREG = a1.l + a1.h, DREG = a0.l + a0.h

Abstract

This instruction adds the accumulator half words together, then it extracts the result to a register. Each half word is sign extended to 32-bits before being added. This operation is used to sum the results of the video [[.:SAD8Vec|Sum of Absolute Differences]] instruction.

See Also (SAD8Vec)

AddAccHalf Description

The dual 16-bit accumulator eExtraction with addition instruction adds together the upper half-words (bits 31through 16) and lower half-words (bits 15 through 0) of each Accumulator and loads each result into a 32-bit destination register.

Each 16-bit half-word in each Accumulator is sign extended before being added together.

A special application of this instruction is to use the dual 16-bit accumulator extraction with addition instruction for motion estimation algorithms in conjunction with the quad 8-bit subtract-absolute-accumulate instruction.

This **16-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

AddAccHalf Example

```
r4=a1.l+a1.h, r7=a0.l+a0.h ;
```

Vectored 8-Bit Sum of Absolute Differences (SAD8Vec)

General Form

Dsp32Alu
saa (PAIR0 , PAIR1) RS

Abstract

This instruction does a vector 8-bit subtract, takes the absolute value of the differences and accumulates them adjusting for byte alignment. The lower two bits of I0 and I1 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair. The four 16-bit results are stored in A1.h, A1.l, A0.h and A0.l These will saturate if the unsigned 16-Bit sections of the accumulator overflow.

See Also (AddAccHalf)

SAD8Vec Description

The quad 8-bit subtract-absolute-accumulate instruction subtracts four pairs of values, takes the absolute value of each difference, and accumulates each result into a 16-bit Accumulator half. The results are placed in the upper- and lower-half Accumulators A0.H, A0.L, A1.H, and A1.L.

Saturation is performed if an operation overflows a 16-bit Accumulator half.

This instruction supports the following byte-wise Sum of Absolute Difference (SAD) calculations.

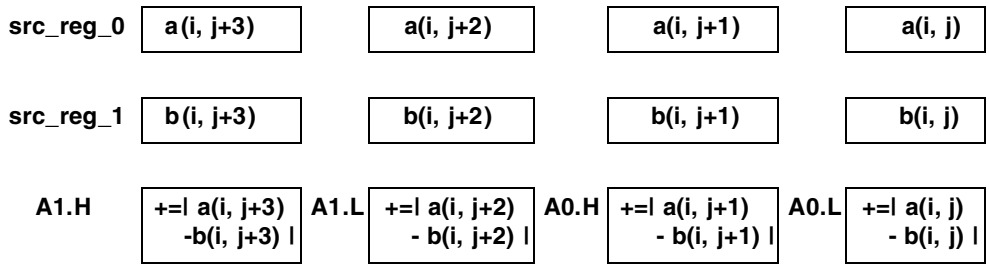
Figure 8-59: Absolute Difference (SAD) Calculations

$$SAD = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a(i,j) - b(i,j)$$

Typical values for N are 8 and 16, corresponding to the video block size of 8x8 and 16x16 pixels, respectively. The 16-bit Accumulator registers limit the pixel region or block size to 32x32 pixels.

The SAA instruction behavior is shown in the **SAA Instruction Behavior** figure.

Figure 8-60: SAA Instruction Behavior



The Quad 8-Bit Subtract-Absolute-Accumulate instruction provides byte alignment directly in the source register pairs src_reg_0 and src_reg_1 based on index registers I0 and I1.

- The two LSBs of the I0 register determine the byte alignment for source register pair src_reg_0 (typically R1:0).
- The two LSBs of the I1 register determine the byte alignment for source register pair src_reg_1 (typically R3:2).

The relationship between the I-register bits and the byte alignment is shown in the **I-register Bits and the Byte Alignment (no reverse)** figure.

In the default source order case (for example, not the (R) syntax), assume a source register pair contain the data shown in the figure.

Figure 8-61: I-register Bits and the Byte Alignment (no reverse)

The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

Options

The (R) syntax reverses the order of the source registers within each pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.

When reversing source order by using the (R) syntax, the source registers swap places within the register pair in their byte ordering. If a source register pair contains the data shown in the **I-register Bits and the Byte Alignment (with reverse)** figure, then the SAA instruction computes 12 pixel operations simultaneously–the three-operation subtract-absolute-accumulate on four pairs of operand bytes in parallel.

Figure 8-62: I-register Bits and the Byte Alignment (with reverse)

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

A special application of this instruction is to use the quad 8-bit subtract-absolute-accumulate instruction for block-based video motion estimation algorithms using block sum of absolute difference (SAD) calculations to measure distortion.

This **16-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

SAD8Vec Example

```
saa (r1:0, r3:2) || r0 = [i0++] || r2 = [i1++] ; /* parallel fill instructions */
saa (r1:0, r3:2) (R) || r1 = [i0++] || r3 = [i1++] ; /* reverse, parallel fill
instructions */
saa (r1:0, r3:2) ; /* last SAA in a loop, no more fill required */
```

Viterbi Operations

These operations provide Viterbi application specific operations on register operands:

- 16-Bit Add on Sign (AddOnSign)
- 16-Bit Modulo Maximum with History (Shift_VitMax)
- Dual 16-Bit Modulo Maximum with History (Shift_DualVitMax)

16-Bit Add on Sign (AddOnSign)

General Form

Dsp32Alu
DREG_H = DREG_L = sign(DREG_H) * DREG_H + sign(DREG_L) * DREG_L

Abstract

This instruction does a vector multiply of the signs SRC0.h and SRC0.l by the values of SRC1.h and SRC1. l Then, it adds the two results and stores it in both of the destination half registers. This instruction does not saturate if the result is greater than 16-Bits.

See Also (Shift_VitMax, Shift_DualVitMax)

AddOnSign Description

The Add on Sign instruction performs a two step function, as follows.

Step 1

Multiply the arithmetic sign of a 16-bit half-word number in src0 by the corresponding half-word number in src1. The arithmetic sign of src0 is either (+1) or (–1), depending on the sign bit of src0. The instruction performs this operation on the upper and lower half-words of the same data registers.

The results of this step obey the signed multiplication rules summarized in the **Signed Multiplication Rules** table. Y is the number in src0, and Z is the number in src1. The numbers in src0 and src1 may be positive or negative. Note the result always bears the magnitude of Z with only the sign affected.

Table 8-45: Signed Multiplication Rules

SRC0	SRC1	Adjusted
+Y	+Z	+Z
+Y	–Z	–Z
–Y	+Z	–Z
–Y	–Z	+Z

Step 2

Add the sign-adjusted src1 upper and lower half-word results together and store the same 16-bit sum in the upper and lower halves of the destination register, as shown in the next figures.

Figure 8-63: Source Registers Contain

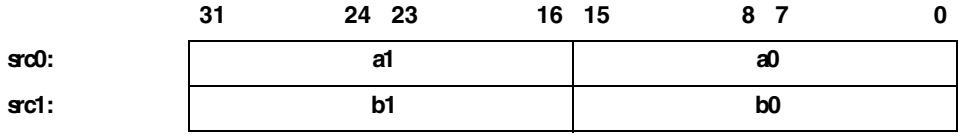
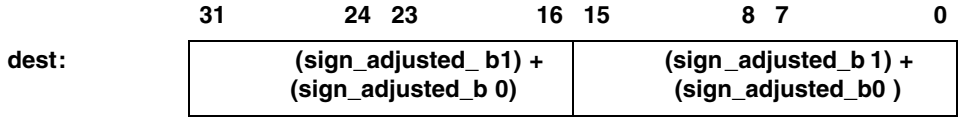


Figure 8-64: Destination Register Receives



The sum is not saturated if the addition exceeds 16 bits.

A special application of this instruction is to use the Sum on Sign instruction to compute the branch metric used by each Viterbi Butterfly.

This **16-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

AddOnSign Example

```
r7.h = r7.l = sign(r2.h) * r3.h + sign(r2.l) * r3.l ;
```

- If

  ```
  R2.H = 2
  R3.H = 23
  R2.L = 2001
  R3.L = 1234
  ```

  then

  ```
  R7.H = 1257 (or 1234 + 23)
  R7.L = 1257
  ```

- If

  ```
  R2.H = -2
  R3.H = 23
  ```

```
R2.L = 2001
R3.L = 1234
```

then

```
R7.H = 1211 (or 1234 - 23)
R7.L = 1211
```

- If

```
R2.H = 2
R3.H = 23
R2.L = -2001
R3.L = 1234
```

then

```
R7.H = -1211 (or (-1234) + 23)
R7.L = -1211
```

- If

```
R2.H = -2
R3.H = 23
R2.L = -2001
R3.L = 1234
```

then

```
R7.H = -1257 (or (-1234) - 23)
R7.L = -1257
```

Dual 16-Bit Modulo Maximum with History (Shift_DualVitMax)

General Form

Dsp32Shf
DREG = vit_max (DREG , DREG) (asl)
DREG = vit_max (DREG , DREG) (asr)

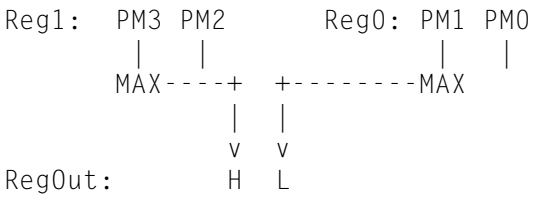
Abstract

This instruction performs maximum value selection and history update. It is used to implement the selection function of a Viterbi decoder. It performs a dual maximum value selection storing the two results in one destination register.

See Also (AddOnSign, Shift_VitMax)

Shift_DualVitMax Description

Maximum value selection and history update. This instruction is used to implement the selection function of a Viterbi decoder. It performs a dual maximum value selection storing the two results in one destination register. In addition shifts left A0 by two bit positions, and stores two bits in A0 representing the result of the two maximum value selections in bit1 and bit0 of A0. No attempt to correct the selection on overflow should be made. This ensures that overflowed path metrics compare correctly, as long as they are close to each other in magnitude.



If the user specifies ASR or ASL this will shift in two bits into the accumulator specifying which 16-bit half register was the max. For ASR it will shift the history bits right, for ASL it will shift them left. To compute the maximum this instruction uses a form of modulo arithmetic where 0x8000 > 0x7fff > 0 > 0xffff > 0x8000.

For more information, see the 16-Bit Modulo Maximum with History (Shift_VitMax) instruction.

Shift_DualVitMax Example

For examples using this instruction, see the 16-Bit Modulo Maximum with History (Shift_VitMax) instruction.

16-Bit Modulo Maximum with History (Shift_ VitMax)

General Form

Dsp32Shf
DREG_L = vit_max (DREG) (asl)
DREG_L = vit_max (DREG) (asr)

Abstract

If the user specifies ASR or ASL, this instruction shifts in a bit into the accumulator specifying which 16-bit half register was the max. For ASR, it will shift the history bits right. For ASL, it will shift them left. To compute the maximum, this instruction uses a form of [[.:modulo_comparisons|modulo arithmetic]] where 0x8000 > 0x7fff > 0 > 0xffff > 0x8000 .

See Also (AddOnSign, Shift_DualVitMax)

Shift_VitMax Description

The Compare-Select (VIT_MAX) instruction selects the maximum values of pairs of 16-bit operands, returns the largest values to the destination register, and serially records in A0.W the source of the maximum.This operation performs signed operations. The operands are compared as two's-complements.

The Accumulator extension bits (bits 39–32) must be cleared before executing this instruction.

Dual 16-Bit Operand Behavior

Versions are available for dual and single 16-bit operations. Whereas the dual versions compare four operands to return two maxima, the single versions compare only two operands to return one maximum.

This operation is illustrated in Table 19-4 and Table 19-5.

Figure 8-65: Source Registers Contain (Dual)

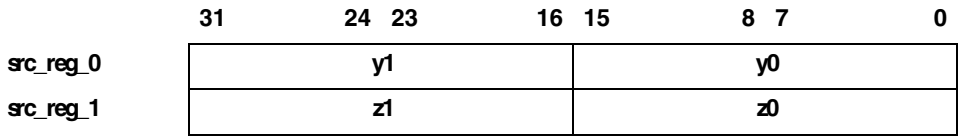
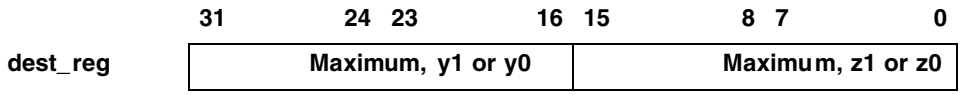


Figure 8-66: Destination Register Contains (Dual)



The ASL version shifts A0 left two bit positions and appends two LSBs to indicate the source of each maximum as shown in Table 19-6 and Table 19-7.

Figure 8-67: ASL Version Shifts (Dual)

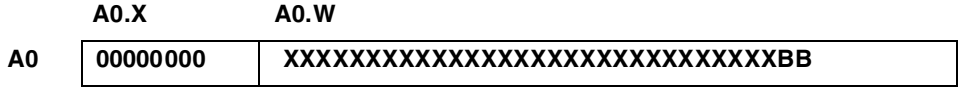


Table 8-46: Where ...

BB	Indicates
00	z0 and y0 are maxima
01	z0 and y1 are maxima
10	z1 and y0 are maxima
11	z1 and y1 are maxima

Conversely, the ASR version shifts A0 right two bit positions and appends two MSBs to indicate the source of each maximum as shown in Table 19-8 and Table 19-9.

Figure 8-68: ASR Version Shifts (Dual)

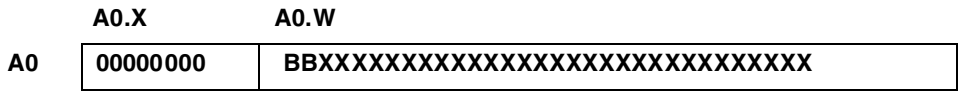


Table 8-47: Where ...

BB	Indicates
00	y0 and z0 are maxima
01	y0 and z1 are maxima
10	y1 and z0 are maxima
11	y1 and z1 are maxima

Notice that the history bit code depends on the A0 shift direction. The bit for src_reg_1 is always shifted onto A0 first, followed by the bit for src_reg_0. The single operand versions behave similarly.

Single 16-Bit Operand Behavior

If the dual source register contains the data shown in Table 19-10 the destination register receives the data shown in Table 19-11.

Figure 8-69: Source Registers Contain (Single)

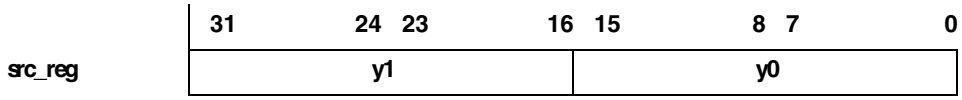
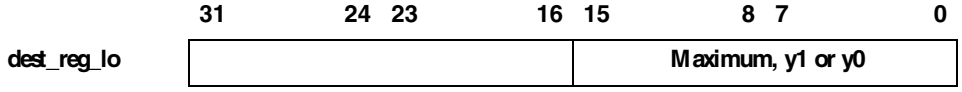
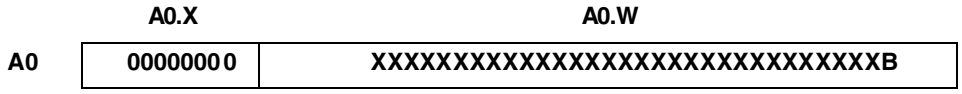


Figure 8-70: Destination Register Contains (Single)



The ASL version shifts A0 left one bit position and appends an LSB to indicate the source of the maximum.

Figure 8-71: ASL Version Shifts (Single)



Conversely, the ASR version shifts A0 right one bit position and appends an MSB to indicate the source of the maximum.

Figure 8-72: ASR Version Shifts (Single)

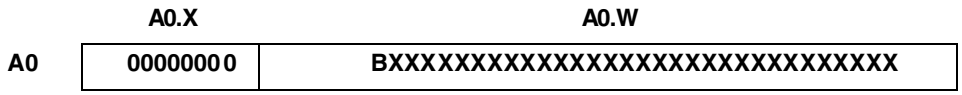


Table 8-48: Where ...

B	Indicates
0	y0 is the maximum
1	y1 is the maximum

The path metrics are allowed to overflow, and maximum comparison is done on the two's-complement circle. Such comparison gives a better indication of the relative magnitude of two large numbers when a small number is added/subtracted to both.

A special application of this instruction is to use the Compare-Select (VIT_MAX) instruction as a key element of the Add-Compare-Select (ACS) function for Viterbi decoders. Combine it with a Vector Add instruction to calculate a trellis butterfly used in ACS functions.

This **32-bit instruction** may be issued in parallel with certain other instructions.

This instruction may be used in either **User or Supervisor mode**.

Shift_VitMax Example

- For:

```
r5 = vit_max(r3, r2)(asl) ; /* shift left, dual operation */
```

Assume:

```
R3 = 0xFFFF 0000
R2 = 0x0000 FFFF
A0 = 0x00 0000 0000
```

This example produces:

```
R5 = 0x0000 0000
A0 = 0x00 0000 0002
```

- For:

```
r7 = vit_max (r1, r0) (asr) ; /* shift right, dual operation */
```

Assume:

```
R1 = 0xFEED BEEF
R0 = 0xDEAF 0000
A0 = 0x00 0000 0000
```

This example produces:

```
R7 = 0xFEED 0000
A0 = 0x00 8000 0000
```

- For:

```
r3.l = vit_max (r1)(asl) ; /* shift left, single operation */
```

Assume:

```
R1 = 0xFFFF 0000
A0 = 0x00 0000 0000
```

This example produces:

```
R3.L = 0x0000
A0 = 0x00 0000 0000
```

- For:

```
r3.l = vit_max (r1)(asr) ; /* shift right, single operation */
```

Assume:

```
R1 = 0x1234 FADE
A0 = 0x00 FFFF FFFF
```

This example produces:

```
R3.L = 0x1234
A0 = 0x00 FFFF FFFF
```

Arithmetic Status Register

The processor updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation. The ASTAT.AQ bit is updated, indicating the status of the most recent Divs or Divq instruction. The ASTAT.RND_MOD bit does not indicate status, rather this bit selects unbiased or biased rounding for operations supporting rounding.

If execution of an instruction generates status, the instruction's reference page indicates the affected arithmetic status bits.

ASTAT: Arithmetic Status Register - R/W

Reset = 0x0000 0000

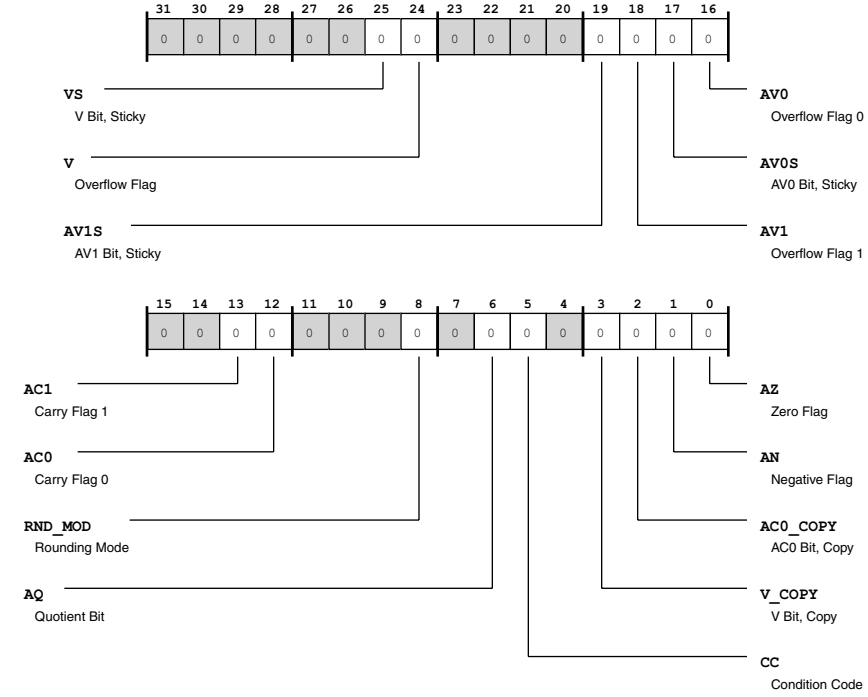


Table 8-49: ASTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
25 (R/W)	VS	V Bit, Sticky. The ASTAT.VS bit is set if ASTAT.VS is set; unaffected otherwise.	
24 (R/W)	V	Overflow Flag. The ASTAT.V bit is set if the most recent ALU0 or MAC0 operation result overflows; cleared if operation generates no overflow,	
19 (R/W)	AV1S	AV1 Bit, Sticky. The ASTAT.AV1S bit is set if ASTAT.AV1 is set; unaffected otherwise.	
18 (R/W)	AV1	Overflow Flag 1. The ASTAT.AV1 bit is set if the most recent MAC1 operation result placed in A1 overflows; cleared if result placed in A1 generates no overflow; sticky for MAC.	
17 (R/W)	AV0S	AV0 Bit, Sticky. The ASTAT.AV0S bit is set if ASTAT.AV0 is set; unaffected otherwise.	
16 (R/W)	AV0	Overflow Flag 0. The ASTAT.AV0 bit is set if the most recent ALU0 or MAC0 operation result placed in A0 overflows; cleared if result placed in A0 generates no overflow; sticky for MAC.	
13 (R/W)	AC1	Carry Flag 1. The ASTAT.AC1 bit is set if the most recent ALU1 operation using A1 generates a carry; cleared if operation using A1 generates no carry.	
12 (R/W)	AC0	Carry Flag 0. The ASTAT.AC0 bit is set if the ALU0 operation using A0 generates a carry; cleared if operation using A0 generates no carry.	
8 (R/W)	RND_MOD	Rounding Mode. The ASTAT.RND_MOD bit is not affected by result status. Instead, this bit is used to select the rounding mode for arithmetic instructions that support rounding.	
		0	Unbiased rounding
		1	Biased rounding
6 (R/W)	AQ	Quotient Bit. (Divs and Divq instructions only) The ASTAT.AQ bit equals the dividend MSB exclusive-OR divisor MSB, where the dividend is a 32-bit value and divisor is a 16-bit value.	
5 (R/W)	CC	Condition Code. (TestSet instruction only) The ASTAT.CC bit is set if the addressed value is zero; cleared if the addressed value is non-zero 0x0.	
3 (R/W)	V_COPY	V Bit, Copy. The ASTAT.V_COPY bit is set if ASTAT.V_COPY is set; cleared if ASTAT.V_COPY is cleared.	
2 (R/W)	AC0_COPY	AC0 Bit, Copy. The ASTAT.AC0_COPY bit is set if ASTAT.AC0 is set; cleared if ASTAT.AC0 is cleared.	

Table 8-49: ASTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
1 (R/W)	AN	Negative Flag. The ASTAT.AN bit is set if the most recent ALU0 or shifter operation result is negative; cleared if result is non-negative.
0 (R/W)	AZ	Zero Flag. The ASTAT.AZ bit is set if the most recent ALU0 or shifter operation result is zero; cleared if result is non-zero.

Instruction Page Tables

The instruction page tables provide definitions of:

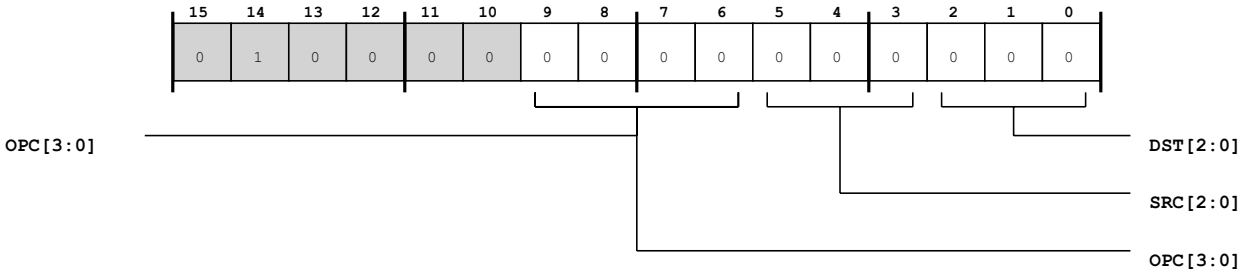
- Instruction types (including opcodes for each instruction)
- Constant types (including immediate value types used in all instructions)
- Register types

These pages are organized alphabetically, with instruction types first, followed by constant types, then register types.

ALU Binary Operations (ALU2op)

ALU2op Instruction Syntax

ALU Binary Operations (ALU2op)



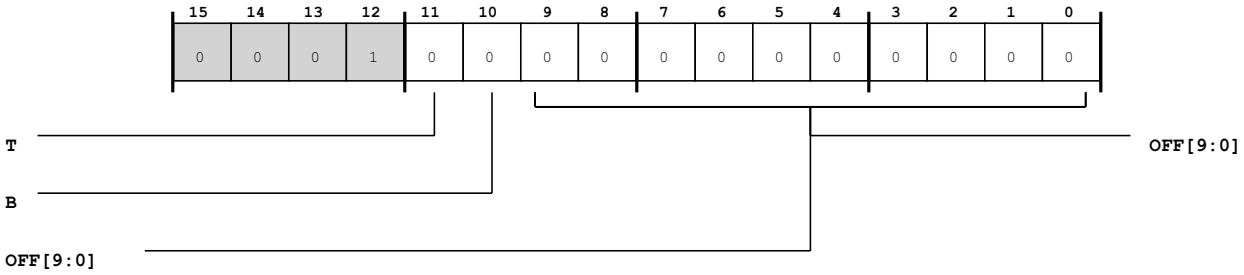
The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
0000	DREG >>>= DREG	AShift32
0001	DREG >>= DREG	LShift
0010	DREG <<= DREG	LShift
0011	DREG *= DREG	MultInt
0100	DREG =(DREG + DREG) << 1	AddSubShift
0101	DREG =(DREG + DREG) << 2	AddSubShift
1000	divq (DREG , DREG)	Divide
1001	divs (DREG , DREG)	Divide
1010	DREG = DREG_L (x)	MvDregLToDreg
1011	DREG = DREG_L (z)	MvDregLToDreg
1100	DREG = DREG_B (x)	MvDregBToDreg
1101	DREG = DREG_B (z)	MvDregBToDreg
1110	DREG =- DREG	Neg32
1111	DREG =~ DREG	Not32

Conditional Branch PC relative on CC (BrCC)

BrCC Instruction Syntax

Conditional Branch PC relative on CC (BrCC)



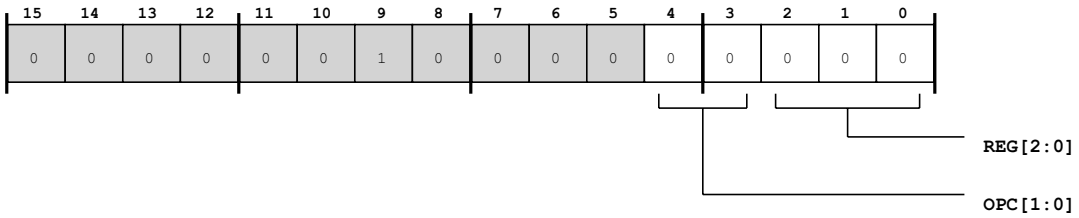
The following table provides the opcode field values (T, B), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

T	B	Syntax	Instruction
0	0	if !cc jump imm10s2	BrCC
0	1	if !cc jump imm10s2 (bp)	BrCC
1	0	if cc jump imm10s2	BrCC
1	1	if cc jump imm10s2 (bp)	BrCC

Move CC conditional bit, to and from dreg (CC2Dreg)

CC2Dreg Instruction Syntax

Move CC conditional bit, to and from dreg (CC2Dreg)



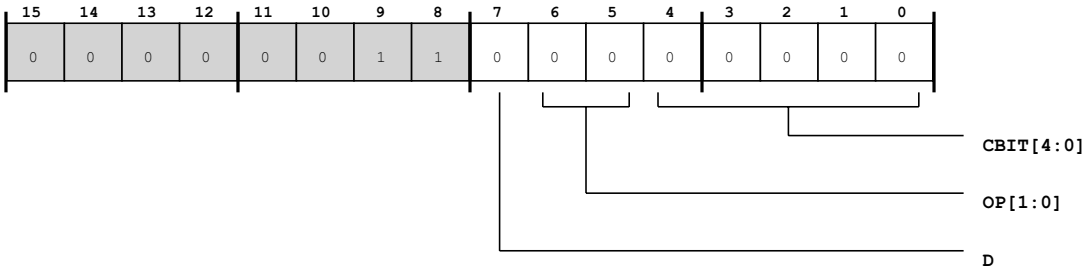
The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction	Rev
00	DREG = cc	CCToDreg	
01	cc = DREG	MvToCC	
10	DREG = !cc	CCToDreg	2.0
11	cc = !cc	MvToCC	

Copy CC conditional bit, from status (CC2Stat)

CC2Stat Instruction Syntax

Copy CC conditional bit, from status (CC2Stat)



The following table provides the opcode field values (D, OP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

D	OP	Syntax	Instruction
0	00	cc = CB	MvToCC_STAT
0	01	cc |= CB	MvToCC_STAT
0	10	cc &= CB	MvToCC_STAT
0	11	cc ^= CB	MvToCC_STAT
1	00	CB = cc	CCToStat16
1	01	CB |= cc	CCToStat16
1	10	CB &= cc	CCToStat16
1	11	CB ^= cc	CCToStat16

CB

CB Encode Table

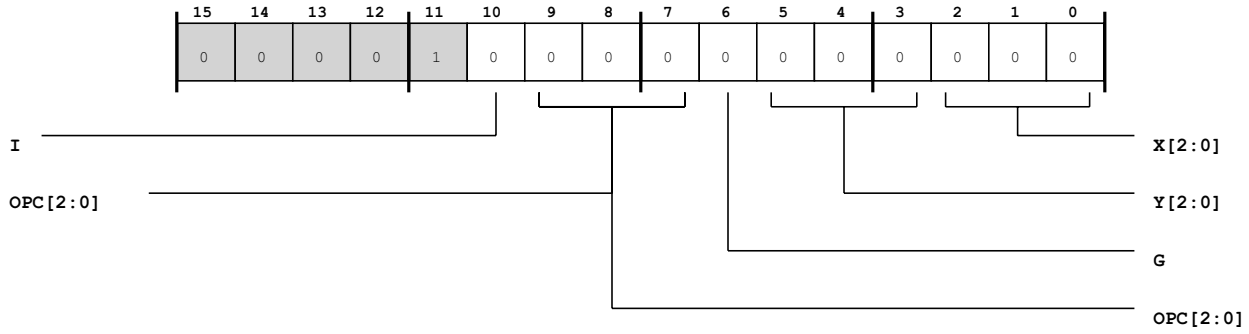
CBIT	Syntax
00000	az
00001	an
00110	aq
01000	rnd_mod

CBIT	Syntax
01100	ac0
01101	ac1
10000	av0
10001	av0s
10010	av1
10011	av1s
11000	v
11001	vs

Set CC conditional bit (CCFlag)

CCFlag Instruction Syntax

Set CC conditional bit (CCFlag)



The following table provides the opcode field values (OPC, G, I), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

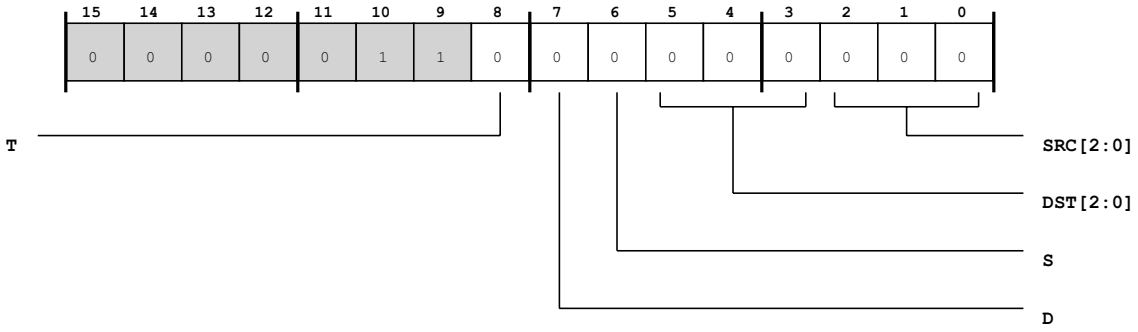
OPC	G	I	Syntax	Instruction
000	0	0	cc = DREG == DREG	CompRegisters
000	0	1	cc = DREG == imm3	CompRegisters
000	1	0	cc = PREG == PREG	CCFlagP
000	1	1	cc = PREG == imm3	CCFlagP
001	0	0	cc = DREG < DREG	CompRegisters
001	0	1	cc = DREG < imm3	CompRegisters
001	1	0	cc = PREG < PREG	CCFlagP
001	1	1	cc = PREG < imm3	CCFlagP
010	0	0	cc = DREG <= DREG	CompRegisters
010	0	1	cc = DREG <= imm3	CompRegisters
010	1	0	cc = PREG <= PREG	CCFlagP
010	1	1	cc = PREG <= imm3	CCFlagP
011	0	0	cc = DREG < DREG (iu)	CompRegisters
011	0	1	cc = DREG < uimm3 (iu)	CompRegisters
011	1	0	cc = PREG < PREG (iu)	CCFlagP
011	1	1	cc = PREG < uimm3 (iu)	CCFlagP
100	0	0	cc = DREG <= DREG (iu)	CompRegisters
100	0	1	cc = DREG <= uimm3 (iu)	CompRegisters

OPC	G	I	Syntax	Instruction
100	1	0	cc = PREG <= PREG (iu)	CCFlagP
100	1	1	cc = PREG <= uimm3 (iu)	CCFlagP
101	0	0	cc = a0 == a1	CompAccumulators
110	0	0	cc = a0 < a1	CompAccumulators
111	0	0	cc = a0 <= a1	CompAccumulators

Conditional Move (CCMV)

CCMV Instruction Syntax

Conditional Move (CCMV)



The following table provides the opcode field values (T), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

T	Syntax	Instruction
1	if cc GDST = GSRC	MvRegToRegCond
0	if !cc GDST = GSRC	MvRegToRegCond

GDST

GDST Encode Table

D	DST	Syntax
0	---	DREG
1	---	PREG

GSRC

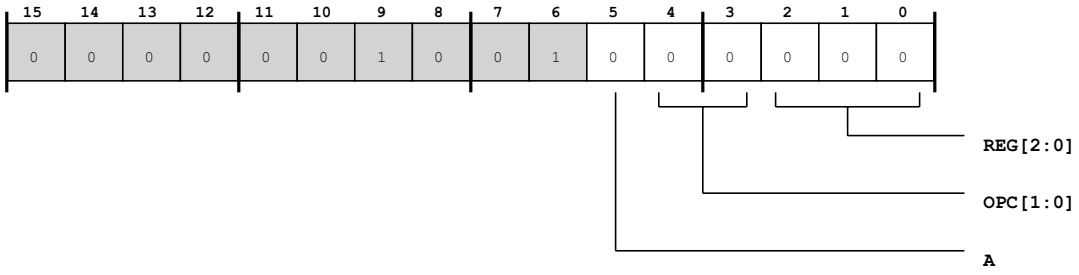
GSRC Encode Table

S	SRC	Syntax
0	---	DREG
1	---	PREG

Cache Control (CacheCtrl)

CacheCtrl Instruction Syntax

Cache Control (CacheCtrl)



The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
00	prefetch [PREGA]	CacheCtrl
01	flushinv [PREGA]	CacheCtrl
10	flush [PREGA]	CacheCtrl
11	iflush [PREGA]	CacheCtrl

PREGA

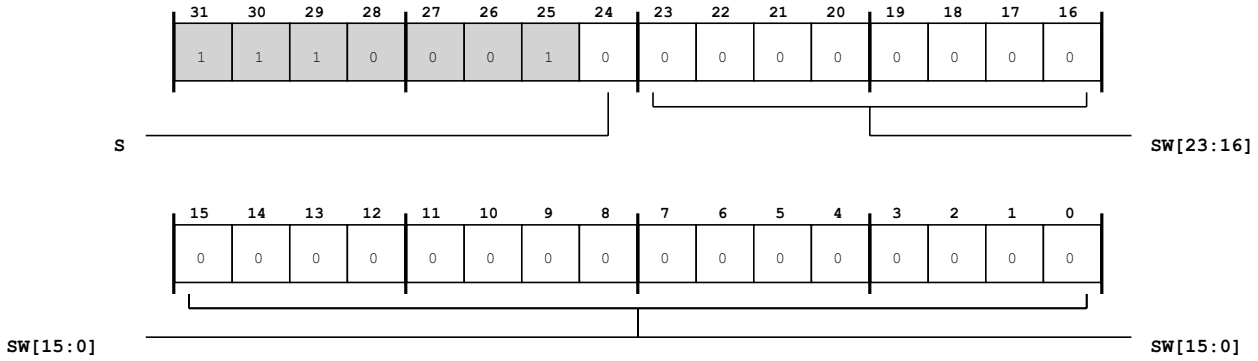
PREGA Encode Table

A	Syntax
0	PREG
1	PREG ++

Call function with pcrel address (CallA)

CallA Instruction Syntax

Call function with pcrel address (CallA)



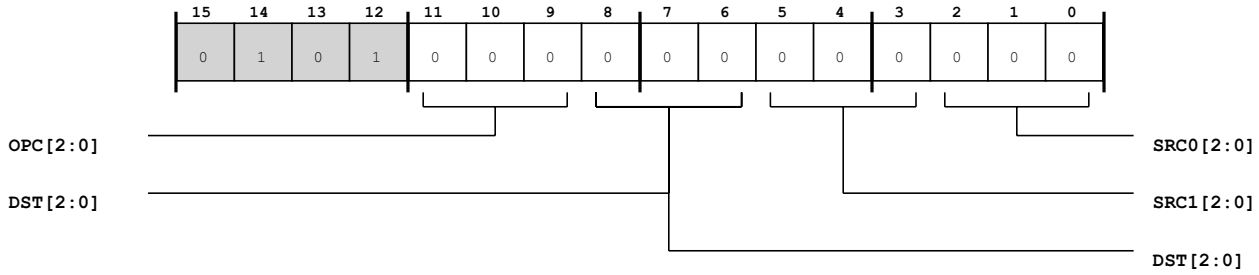
The following table provides the opcode field values (S), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

S	Syntax	Instruction
0	jump.l imm24s2	JumpAbs
1	call imm24nxs2	Call

Compute with 3 operands (Comp3op)

Comp3op Instruction Syntax

Compute with 3 operands (Comp3op)



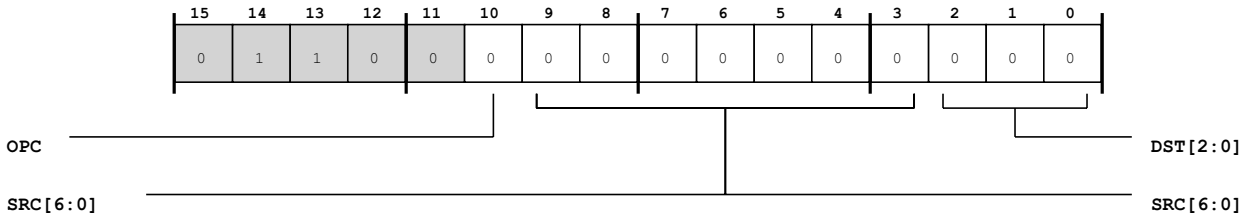
The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
000	DREG = DREG + DREG	AddSub32
001	DREG = DREG - DREG	AddSub32
010	DREG = DREG & DREG	Logic32
011	DREG = DREG | DREG	Logic32
100	DREG = DREG ^ DREG	Logic32
101	PREG = PREG + PREG	DagAdd32
110	PREG = PREG +(PREG << 1)	PtrOp
111	PREG = PREG +(PREG << 2)	PtrOp

Destructive Binary Operations, dreg with 7bit immediate (CompI2opD)

CompI2opD Instruction Syntax

Destructive Binary Operations, dreg with 7bit immediate (CompI2opD)



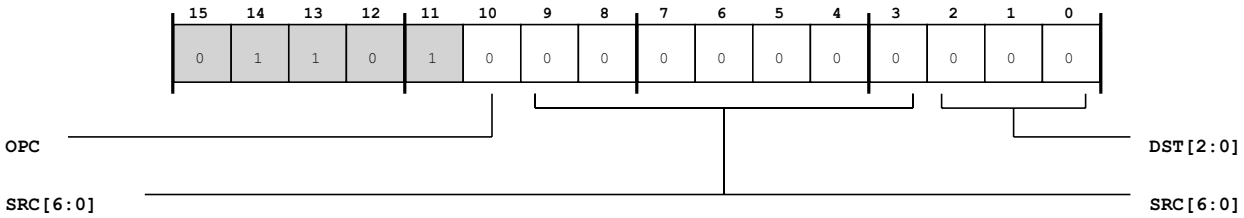
The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
0	DREG = imm7 (x)	LdImmToReg
1	DREG += imm7	AddImm

Destructive Binary Operations, preg with 7bit immediate (Compl2opP)

Compl2opP Instruction Syntax

Destructive Binary Operations, preg with 7bit immediate (Compl2opP)



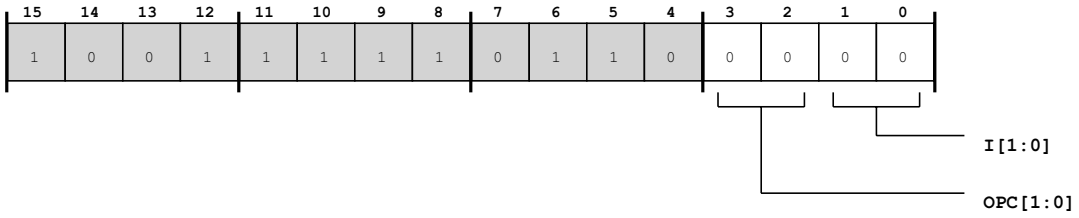
The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
0	PREG = imm7 (x)	LdImmToReg
1	PREG += imm7	DagAddImm

DAG Arithmetic (DAGModIk)

DAGModIk Instruction Syntax

DAG Arithmetic (DAGModIk)



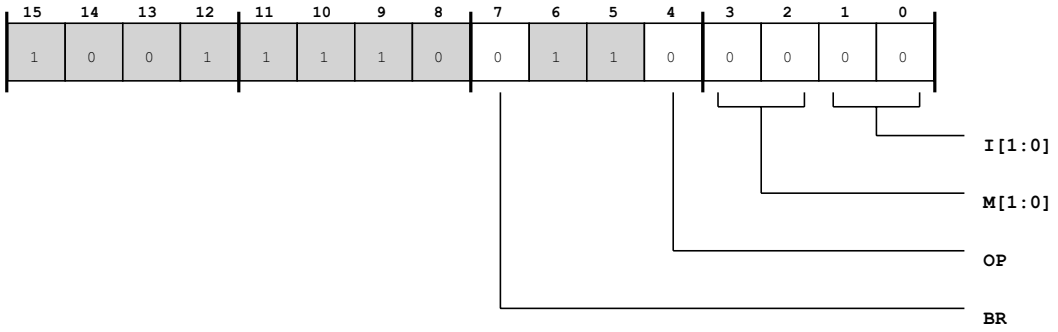
The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
00	IREG += 2	DagAddImm
01	IREG -= 2	DagAddImm
10	IREG += 4	DagAddImm
11	IREG -= 4	DagAddImm

DAG Arithmetic (DAGModIm)

DAGModIm Instruction Syntax

DAG Arithmetic (DAGModIm)



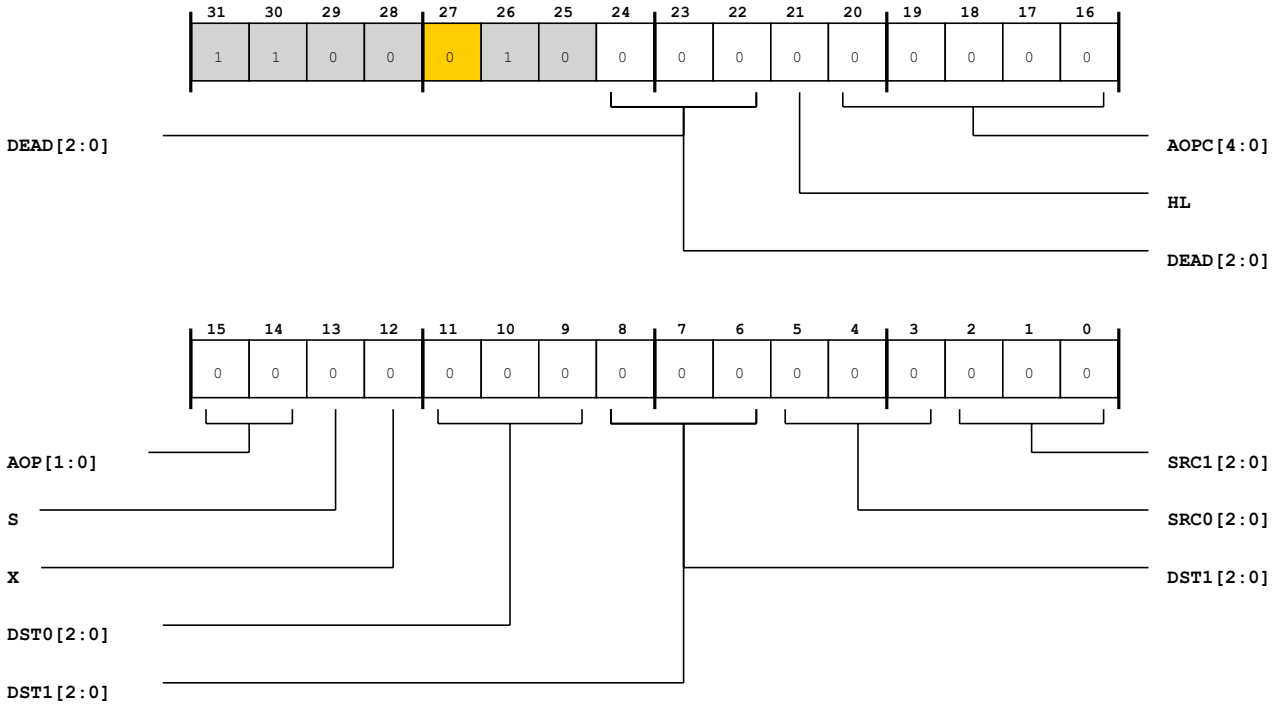
The following table provides the opcode field values (OP, BR), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OP	BR	Syntax	Instruction
0	0	IREG += MREG	DagAdd32
0	1	IREG += MREG (brev)	DagAdd32
1	0	IREG -= MREG	DagAdd32

ALU Operations (Dsp32Alu)

Dsp32Alu Instruction Syntax

ALU Operations (Dsp32Alu)



The following table provides the opcode field values (AOPC, AOP, HL, S, X), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

AOPC	AOP	HL	S	X	Syntax	Instruction	Rev
00000	--	-	-	-	DREG = DREG AOPL DREG SX	AddSubVec16	
00001	--	0	-	-	DREG = DREG +|+ DREG , DREG = DREG -|- DREG SXA	AddSubVec16	
00001	--	1	-	-	DREG = DREG +|- DREG , DREG = DREG -|+ DREG SXA	AddSubVec16	
00010	00	-	-	0	DDSTO_HL = DREG_L + DREG_L SAT2	AddSub16	
00010	01	-	-	0	DDSTO_HL = DREG_L + DREG_H SAT2	AddSub16	
00010	10	-	-	0	DDSTO_HL = DREG_H + DREG_L SAT2	AddSub16	
00010	11	-	-	0	DDSTO_HL = DREG_H + DREG_H SAT2	AddSub16	
00011	00	-	-	0	DDSTO_HL = DREG_L - DREG_L SAT2	AddSub16	
00011	01	-	-	0	DDSTO_HL = DREG_L - DREG_H SAT2	AddSub16	

AOPC	AOP	HL	S	X	Syntax	Instruction	Rev
00011	10	-	-	0	DDSTO_HL = DREG_H - DREG_L SAT2	AddSub16	
00011	11	-	-	0	DDSTO_HL = DREG_H - DREG_H SAT2	AddSub16	
00100	00	0	-	0	DREG = DREG + DREG NSAT	AddSub32	
00100	01	0	-	0	DREG = DREG - DREG NSAT	AddSub32	
00100	10	0	-	0	DREG = DREG + DREG , DREG = DREG - DREG SAT	AddSub32Dual	
00101	00	-	0	0	DDSTO_HL = DREG + DREG (rnd12)	AddSubRnd12	
00101	01	-	0	0	DDSTO_HL = DREG - DREG (rnd12)	AddSubRnd12	
00101	10	-	0	1	DDSTO_HL = DREG + DREG (rnd20)	AddSubRnd20	
00101	11	-	0	1	DDSTO_HL = DREG - DREG (rnd20)	AddSubRnd20	
00110	00	0	0	0	DREG = max(DREG , DREG) (v)	Max16Vec	
00110	01	0	0	0	DREG = min(DREG , DREG) (v)	Min16Vec	
00110	10	0	0	0	DREG = abs DREG (v)	Abs2x16	
00111	00	0	0	0	DREG = max(DREG , DREG)	Max32	
00111	01	0	0	0	DREG = min(DREG , DREG)	Min32	
00111	10	0	0	0	DREG = abs DREG	Abs32	
00111	11	0	-	0	DREG = - DREG NSAT	Neg32	
01000	00	0	0	0	a0 = 0	LdImmToAx	
01000	00	0	1	0	a0 = a0 (s)	ALU_SatAcc0	
01000	01	0	0	0	a1 = 0	LdImmToAx	
01000	01	0	1	0	a1 = a1 (s)	ALU_SatAcc1	
01000	10	0	0	0	a1 = a0 = 0	LdImmToAxDual	
01000	10	0	1	0	a1 = a1 (s), a0 = a0 (s)	ALU_SatAccDual	
01000	11	0	0	0	a0 = a1	MvAxToAx	
01000	11	0	1	0	a1 = a0	MvAxToAx	
01001	00	-	0	0	A0_HL = DSRC0_HL	MvDregHLToAxHL	
01001	00	0	1	-	a0 = DREG XMODE	MvDregToAx	
01001	01	0	0	0	a0.x = DREG_L	MvDregLToAxX	
01001	10	-	0	0	A1_HL = DSRC0_HL	MvDregHLToAxHL	
01001	10	0	1	-	a1 = DREG XMODE	MvDregToAx	
01001	11	0	0	0	a1.x = DREG_L	MvDregLToAxX	
01010	00	0	0	0	DREG_L = a0.x	MvAxXToDregL	
01010	01	0	0	0	DREG_L = a1.x	MvAxXToDregL	
01011	00	0	0	0	DREG = (a0 += a1)	AddAccExt	
01011	01	-	0	0	DDSTO_HL = (a0 += a1)	AddAccExt	
01011	10	0	0	0	a0 += a1	AddSubAcc	

AOPC	AOP	HL	S	X	Syntax	Instruction	Rev
01011	10	0	1	0	a0 += a1 (w32)	AddSubAcc	
01011	11	0	0	0	a0 -= a1	AddSubAcc	
01011	11	0	1	0	a0 -= a1 (w32)	AddSubAcc	
01100	00	0	0	0	DREG_H = DREG_L = sign(DREG_H)* DREG_H + sign(DREG_L)* DREG_L	AddOnSign	
01100	01	0	0	0	DREG = a1.l + a1.h, DREG = a0.l + a0.h	AddAccHalf	
01100	11	-	0	0	DDSTO_HL = DREG (rnd)	Pass32Rnd16	
01101	00	0	0	0	(DREG , DREG) = search DREG (gt)	Search	
01101	01	0	0	0	(DREG , DREG) = search DREG (ge)	Search	
01101	10	0	0	0	(DREG , DREG) = search DREG (lt)	Search	
01101	11	0	0	0	(DREG , DREG) = search DREG (le)	Search	
01110	00	0	0	0	a0 = -a0	NegAcc0	
01110	00	1	0	0	a1 = -a0	NegAcc1	
01110	01	0	0	0	a0 = -a1	NegAcc0	
01110	01	1	0	0	a1 = -a1	NegAcc1	
01110	11	0	0	0	a1 = -a1, a0 = -a0	NegAccDual	
01111	11	0	0	0	DREG = - DREG (v)	Neg16Vec	
10000	00	0	0	0	a0 = abs a0	AbsAcc0	
10000	00	1	0	0	a1 = abs a0	AbsAcc1	
10000	01	0	0	0	a0 = abs a1	AbsAcc0	
10000	01	1	0	0	a1 = abs a1	AbsAcc1	
10000	11	0	0	0	a1 = abs a1, a0 = abs a0	AbsAccDual	
10000	11	1	-	-	a1 = DREG SMODE , a0 = DREG XMODE	MvDregToAxDual	2.1
10001	00	0	-	0	DREG = a1 + a0, DREG = a1 - a0 SAT	AddSubAccExt	
10001	01	0	-	0	DREG = a0 + a1, DREG = a0 - a1 SAT	AddSubAccExt	
10010	00	0	-	0	saa (PAIR0 , PAIR1) RS	SAD8Vec	
10010	11	0	0	0	disalgnexcpt	DisAlignExcept	
10100	00	0	-	0	DREG = byteop1p (PAIR0 , PAIR1) RS	Avg8Vec	
10100	01	0	-	0	DREG = byteop1p (PAIR0 , PAIR1) (t RSC)	Avg8Vec	
10101	00	0	-	0	(DREG , DREG) = byteop16p (PAIR0 , PAIR1) RS	AddSub4x8	
10101	01	0	-	0	(DREG , DREG) = byteop16m (PAIR0 , PAIR1) RS	AddSub4x8	
10110	00	0	-	0	DREG = byteop2p (PAIR0 , PAIR1) (rndl RSC)	Avg4x8Vec	
10110	00	1	-	0	DREG = byteop2p (PAIR0 , PAIR1) (rndh RSC)	Avg4x8Vec	
10110	01	0	-	0	DREG = byteop2p (PAIR0 , PAIR1) (tl RSC)	Avg4x8Vec	
10110	01	1	-	0	DREG = byteop2p (PAIR0 , PAIR1) (th RSC)	Avg4x8Vec	
10111	00	0	-	0	DREG = byteop3p (PAIR0 , PAIR1) (lo RSC)	AddClip	

AOPC	AOP	HL	S	X	Syntax	Instruction	Rev
10111	00	1	-	0	DREG = byteop3p (PAIR0 , PAIR1) (hi RSC)	AddClip	
11000	00	0	0	0	DREG = bytepack (DREG , DREG)	BytePack	
11000	01	0	-	0	(DREG , DREG) = byteunpack PAIR0 RS	ByteUnPack	
11001	00	0	-	0	DREG = DREG + DREG + ac0 SAT	AddSubAC0	2.0
11001	01	0	-	0	DREG = DREG - DREG + ac0 - 1 SAT	AddSubAC0	2.0

A0_HL

A0_HL Encode Table

HL	Syntax
0	a0.l
1	a0.h

A1_HL

A1_HL Encode Table

HL	Syntax
0	a1.l
1	a1.h

AOPL

AOPL Encode Table

AOP	Syntax
00	+|+
01	+|-
10	-|+
11	-|-

DDST0_HL

DDST0_HL Encode Table

HL	Syntax
0	DREG_L
1	DREG_H

DSRC0_HL

DSRC0_HL Encode Table

HL	Syntax
0	DREG_L
1	DREG_H

NSAT

NSAT Encode Table

S	Syntax	Description
0	(ns)	The (ns) option directs the ALU not to saturate the result.
1	(s)	The (s) option directs the ALU to saturate the result at 16 or 32 bits, depending on the operand size.

PAIR0

PAIR0 Encode Table

SRC0	Syntax
00-	r1:0
01-	r3:2

PAIR1

PAIR1 Encode Table

SRC1	Syntax
00-	r1:0
01-	r3:2

RS

RS Encode Table

S	Syntax	Description
0		The default (no option select) operation directs the ALU to execute the operation with no optional modification to the result.
1	(r)	The `(r)` option directs the ALU to reverse the order of the source registers within each register pair.

RSC

RSC Encode Table

S	Syntax	Description
0		The default (no option select) operation directs the ALU to execute the operation with no optional modification to the result.
1	,r	The `,r` option directs the ALU to reverse the order of the source registers within each register pair.

SAT

SAT Encode Table

S	Syntax	Description
0	(ns)	The `(ns)` option directs the ALU not to saturate the result.
1	(s)	The `(s)` option directs the ALU to saturate the result at 16 or 32 bits, depending on the operand size.

SAT2

SAT2 Encode Table

S	Syntax	Description
0	(ns)	The (ns) option directs the ALU not to saturate the result.
1	(s)	The (s) option directs the ALU to saturate the result at 16 or 32 bits, depending on the operand size.

SMODE

SMODE Encode Table

S	Syntax	Description	Rev
0	(x)	The (x) option directs the ALU to sign-extend the result.	2.1
1	(z)	The (z) option directs the ALU to zero extend the result.	2.1

SX

SX Encode Table

S	X	Syntax	Description
0	0		The default (no option select) operation directs the ALU to execute the operation with no optional modification to the result.
0	1	(co)	The (co) option directs the ALU to swap (cross order) the order of the results in the destination register.
1	0	(s)	The (s) option directs the ALU to saturate the result at 16 or 32 bits, depending on the operand size.
1	1	(sco)	The (sco) option directs the ALU to apply the combination of the (co) and (s) options.

SXA

SXA Encode Table

AOP	S	X	Syntax	Description
00	0	0		The default (no option select) operation directs the ALU to execute the operation with no optional modification to the result.
00	0	1	(co)	The `(co)` option directs the ALU to swap (cross order) the order of the results in the destination register.
00	1	0	(s)	The `(s)` option directs the ALU to saturate the result at 16 or 32 bits, depending on the operand size.
00	1	1	(sco)	The `(sco)` option directs the ALU to apply the combination of the `(co)` and `(s)` options.
10	0	0	(asr)	The `(asr)` option directs the ALU to arithmetic shift right, halving the result (divide by 2) before storing in the destination register.
10	0	1	(co, asr)	The `(co, asr)` option directs the ALU to to apply the combination of the `(asr)` and `(co)` options
10	1	0	(s, asr)	The `(s, asr)` option directs the ALU to arithmetic shift right (halving the result; divide by 2) then saturate before storing in the destination register.
10	1	1	(sco, asr)	The `(sco, asr)` option directs the ALU to apply the combination of the `(asr)` and `(sco)` options
11	0	0	(asl)	The `(asl)` option directs the ALU to arithmetic shift left, doubling the result (multiply by 2, truncated) before storing in the destination register.
11	0	1	(co, asl)	The `(co, asl)` option directs the ALU to to apply the combination of the `(asl)` and `(co)` options
11	1	0	(s, asl)	The `(s, asl)` option directs the ALU to arithmetic shift left (doubling the result; multiply by 2, truncated) then saturate before storing in the destination register.
11	1	1	(sco, asl)	The `(sco, asl)` option directs the ALU to apply the combination of the `(asl)` and `(sco)` options

XMODE

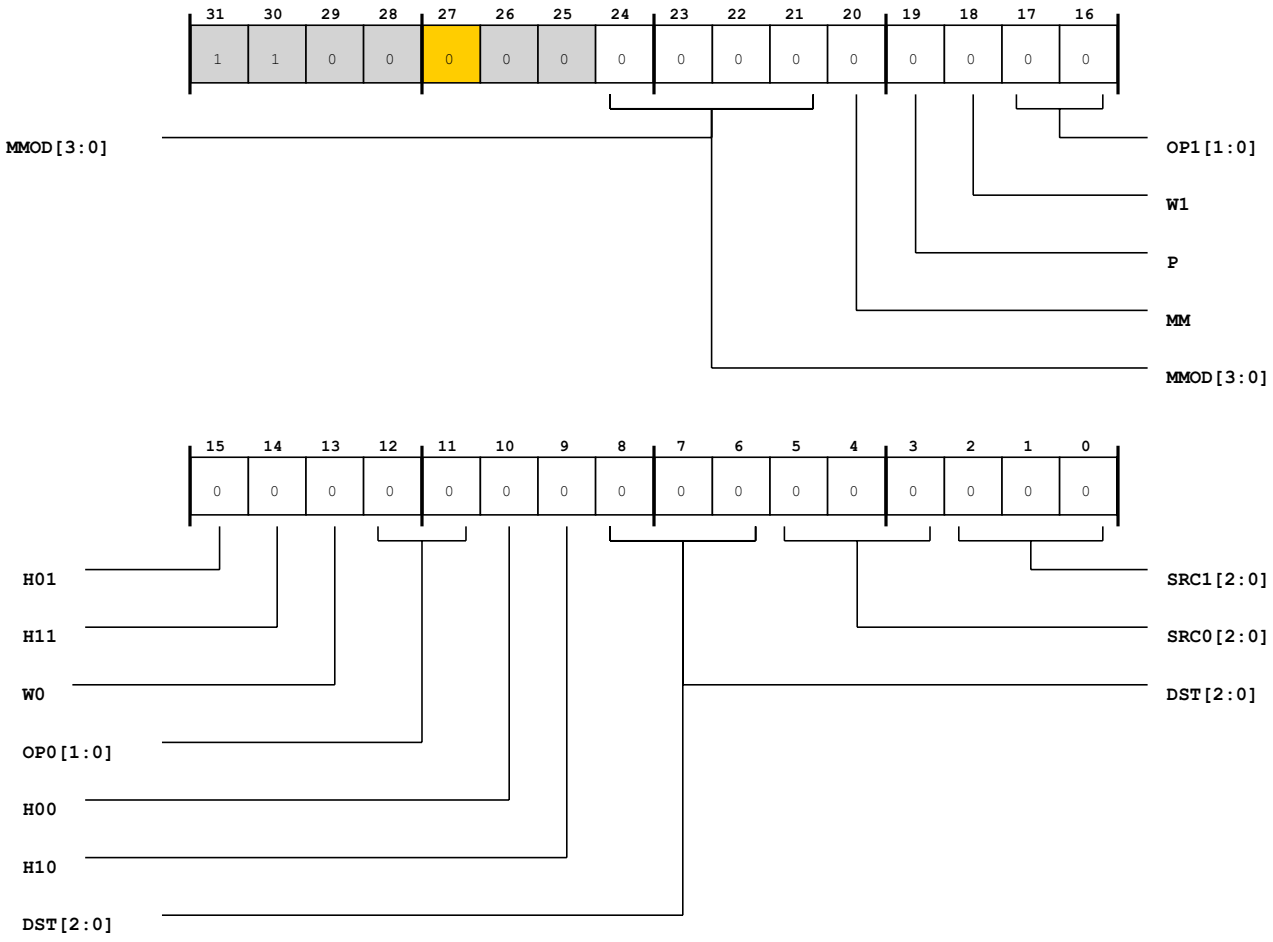
XMODE Encode Table

X	Syntax	Description	Rev
0	(x)	The `(x)` option directs the ALU to sign-extend the result.	2.1
1	(z)	The `(z)` option directs the ALU to zero extend the result.	2.1

Multiply Accumulate (Dsp32Mac)

Dsp32Mac Instruction Syntax

Mulitply Accumulate (Dsp32Mac)



The following table provides the opcode field values (MMOD), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

MMOD	Syntax
0---	TRADMAC
10--	TRADMAC
1100	TRADMAC
1101	CMPLXMAC
111-	CMPLXMAC

CMODE

CMODE Encode Table

MMOD	Syntax	Description
1101		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1111	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

CMPLXMAC

CMPLXMAC Encode Table

W0	P	OP0	Syntax Instruction	Rev
0	0	00	a1:0 = `CMPLXOP CMODE`	2.0
0	0	01	a1:0 += `CMPLXOP CMODE`	2.0
0	0	10	a1:0 -= `CMPLXOP CMODE`	2.0
1	0	00	`DREG` = (a1:0 = `CMPLXOP`) `NARROWING_CMODE`	2.0
1	0	01	`DREG` = (a1:0 += `CMPLXOP`) `NARROWING_CMODE`	2.0
1	0	10	`DREG` = (a1:0 -= `CMPLXOP`) `NARROWING_CMODE`	2.0
1	0	11	`DREG` = `CMPLXOP NARROWING_CMODE`	2.0
1	1	00	`DREG_PAIR` = (a1:0 = `CMPLXOP`) `CMODE`	2.0

W0	P	OP0	Syntax Instruction	Rev
1	1	01	DREG_PAIR = (a1:0 += CMPLXOP) CMODE	2.0
1	1	10	DREG_PAIR = (a1:0 -= CMPLXOP) CMODE	2.0
1	1	11	DREG_PAIR = CMPLXOP CMODE	2.0

CMPLXOP

CMPLXOP Encode Table

OP1	Syntax
00	cmul(DREG , DREG)
01	cmul(DREG , DREG *)
10	cmul(DREG *, DREG *)

MAC0

MAC0 Encode Table

OP0	Syntax
00	a0 = MACOS
01	a0 += MACOS
10	a0 -= MACOS

MAC0S

MAC0S Encode Table

H00	H10	Syntax
0	0	DREG_L * DREG_L
0	1	DREG_L * DREG_H
1	0	DREG_H * DREG_L
1	1	DREG_H * DREG_H

MAC1

MAC1 Encode Table

OP1	Syntax
00	a1 = MAC1S
01	a1 += MAC1S
10	a1 -= MAC1S

MAC1S

MAC1S Encode Table

H01	H11	Syntax
0	0	DREG_L * DREG_L
0	1	DREG_L * DREG_H
1	0	DREG_H * DREG_L
1	1	DREG_H * DREG_H

MML

MML Encode Table

MM	Syntax	Description
0		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round Accumulator 9.31 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MM	Syntax	Description
1	(m)	The (m) option directs the MAC to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *Src_reg_0* is the signed operand and *Src_reg_1* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.

MMLMMOD0

MMLMMOD0 Encode Table

MM	MMOD	Syntax	Description
0	0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round Accumulator 9.31 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	0011	(w32)	The (w32) option directs the MAC to use signed fraction with 32-bit saturation. Multiply 1.15 x 1.15 to produce 1.31 format data after shift correction. Sign extend the result to 9.31 format before passing it to the accumulator. Saturate the accumulator after copying or accumulating at bit 31 to maintain 1.31 precision. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0xFF 8000 0000 and maximum 0x00 7FFF FFFF).
0	0100	(fu)	The (fu) option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).

MM	MMOD	Syntax	Description
0	1000	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1	0000	(m)	The `(m)` option directs the MAC to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *Src_reg_0* is the signed operand and *Src_reg_1* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1	0011	(m,w32)	The `(m,w32)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed fraction (with 32-bit saturation) operation.
1	0100	(m,fu)	The `(m,fu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format operation.
1	1000	(m,is)	The `(m,is)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.

MMLMMOD1

MMLMMOD1 Encode Table

MM	MMOD	Syntax	Description
0	0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1. 15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MM	MMOD	Syntax	Description
0	0001	(s2rnd)	The `(s2rnd)` option directs the MAC to use signed fraction format with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF)
0	0010	(t)	The `(t)` option directs the MAC to use signed fraction format with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0	0100	(fu)	The `(fu)` option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0	0110	(tfu)	The `(tfu)` option directs the MAC to use unsigned fraction format with truncation. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate Accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).

MM	MMOD	Syntax	Description
0	1000	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1001	(iss2)	The `(iss2)` option directs the MAC to use signed integer format with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1011	(ih)	The `(ih)` option directs the MAC to use signed integer format, high word extract. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to Accumulator. Then, saturate accumulator to maintain 32.0 precision; accumulator result is between minimum 0x00 8000 0000 and maximum 0x00 7FFF FFFF. To extract to half-register, round accumulator 40.0 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate to 32.0 result. Copy the upper 16 bits of that value to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0	1100	(iu)	The `(iu)` option directs the MAC to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum $2^{16}-1$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
1	0000	(m)	The `(m)` option directs the MAC to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1	0001	(m,s2rnd)	The `(m,s2rnd)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with scaling and rounding) operation.

MM	MMOD	Syntax	Description
1	0010	(m,t)	The `(m,t)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with truncation) operation.
1	0100	(m,fu)	The `(m,fu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format operation.
1	0110	(m,tfu)	The `(m,tfu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format (with truncation) operation.
1	1000	(m,is)	The `(m,is)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1	1001	(m,iss2)	The `(m,iss2)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format with scaling operation.
1	1011	(m,ih)	The `(m,ih)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format (high word extract) operation.
1	1100	(m,iu)	The `(m,iu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned integer format operation.

MMLMMODE

MMLMMODE Encode Table

MM	MMOD	Syntax	Description
0	0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MM	MMOD	Syntax	Description
0	0001	(s2rnd)	The `(s2rnd)` option directs the MAC to use signed fraction format with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF)
0	0100	(fu)	The `(fu)` option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0	1000	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MM	MMOD	Syntax	Description
0	1001	(iss2)	The `(iss2)` option directs the MAC to use signed integer format with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1100	(iu)	The `(iu)` option directs the MAC to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum $2^{16}-1$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
1	0000	(m)	The `(m)` option directs the MAC to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *Src_reg_0* is the signed operand and *Src_reg_1* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1	0001	(m,s2rnd)	The `(m,s2rnd)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with scaling and rounding) operation.
1	0100	(m,fu)	The `(m,fu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format operation.
1	1000	(m,is)	The `(m,is)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1	1001	(m,iss2)	The `(m,iss2)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format with scaling operation.
1	1100	(m,iu)	The `(m,iu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned integer format operation.

MMOD0

MMOD0 Encode Table

MMOD	Syntax	Description
0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round Accumulator 9.31 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0011	(w32)	The (w32) option directs the MAC to use signed fraction with 32-bit saturation. Multiply 1.15 x 1.15 to produce 1.31 format data after shift correction. Sign extend the result to 9.31 format before passing it to the accumulator. Saturate the accumulator after copying or accumulating at bit 31 to maintain 1.31 precision. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0xFF 8000 0000 and maximum 0x00 7FFF FFFF).
0100	(fu)	The (fu) option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
1000	(is)	The (is) option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MMOD1

MMOD1 Encode Table

MMOD	Syntax	Description
0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0001	(s2rnd)	The `(s2rnd)` option directs the MAC to use signed fraction format with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF)
0010	(t)	The `(t)` option directs the MAC to use signed fraction format with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).

MMOD	Syntax	Description
0100	(fu)	The (fu) option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1-2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1-2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0110	(tfu)	The (tfu) option directs the MAC to use unsigned fraction format with truncation. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate Accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1-2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
1000	(is)	The (is) option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1001	(iss2)	The (iss2) option directs the MAC to use signed integer format with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MMOD	Syntax	Description
1011	(ih)	The (ih) option directs the MAC to use signed integer format, high word extract. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to Accumulator. Then, saturate accumulator to maintain 32.0 precision; accumulator result is between minimum 0x00 8000 0000 and maximum 0x00 7FFF FFFF. To extract to half-register, round accumulator 40.0 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate to 32.0 result. Copy the upper 16 bits of that value to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
1100	(iu)	The (iu) option directs the MAC to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum $2^{16}-1$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).

MMODE

MMODE Encode Table

MMOD	Syntax	Description
0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round Accumulator 9.31 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MMOD	Syntax	Description
0001	(s2rnd)	The (s2rnd) option directs the MAC to use signed fraction format with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF)
0100	(fu)	The (fu) option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1-2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1-2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
1000	(is)	The (is) option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1001	(iss2)	The (iss2) option directs the MAC to use signed integer format with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MMOD	Syntax	Description
1100	(iu)	The (iu) option directs the MAC to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum 2^{16}-1 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).

NARROWING_CMODE

NARROWING_CMODE Encode Table

MMOD	Syntax	Description
1101		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round Accumulator 9.31 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1110	(t)	The (t) option directs the MAC to use signed fraction format with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
1111	(is)	The (is) option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

TRADMAC

TRADMAC Encode Table

P	W1	W0	OP1	OP0	Syntax	Instruction
0	0	0	11	0-	MAC0 MMOD0	Mac16
0	0	0	11	10	MAC0 MMOD0	Mac16
0	0	0	0-	11	MAC1 MMLMMOD0	Mac16
0	0	0	10	11	MAC1 MMLMMOD0	Mac16
0	0	0	0-	0-	MAC1 MML , MAC0 MMOD0	ParaMac16AndMac16
0	0	0	0-	10	MAC1 MML , MAC0 MMOD0	ParaMac16AndMac16
0	0	0	10	0-	MAC1 MML , MAC0 MMOD0	ParaMac16AndMac16
0	0	0	10	10	MAC1 MML , MAC0 MMOD0	ParaMac16AndMac16
0	0	1	11	11	DREG_L = a0 MMOD1	MvA0ToDregL
0	0	1	11	0-	DREG_L = (MAC0) MMOD1	Mac16WithMv
0	0	1	11	10	DREG_L = (MAC0) MMOD1	Mac16WithMv
0	0	1	0-	11	MAC1 MML , DREG_L = a0 MMOD1	ParaMac16AndMv
0	0	1	10	11	MAC1 MML , DREG_L = a0 MMOD1	ParaMac16AndMv
0	0	1	0-	0-	MAC1 MML , DREG_L = (MAC0) MMOD1	ParaMac16AndMac16WithMv
0	0	1	0-	10	MAC1 MML , DREG_L = (MAC0) MMOD1	ParaMac16AndMac16WithMv
0	0	1	10	0-	MAC1 MML , DREG_L = (MAC0) MMOD1	ParaMac16AndMac16WithMv
0	0	1	10	10	MAC1 MML , DREG_L = (MAC0) MMOD1	ParaMac16AndMac16WithMv
0	1	0	11	11	DREG_H = a1 MMLMMOD1	MvA1ToDregH
0	1	0	11	0-	DREG_H = a1 MML , MAC0 MMOD1	ParaMvAndMac16
0	1	0	11	10	DREG_H = a1 MML , MAC0 MMOD1	ParaMvAndMac16
0	1	0	0-	11	DREG_H = (MAC1) MMLMMOD1	Mac16WithMv
0	1	0	10	11	DREG_H = (MAC1) MMLMMOD1	Mac16WithMv
0	1	0	0-	0-	DREG_H = (MAC1) MML , MAC0 MMOD1	ParaMac16WithMvAndMac16
0	1	0	0-	10	DREG_H = (MAC1) MML , MAC0 MMOD1	ParaMac16WithMvAndMac16
0	1	0	10	0-	DREG_H = (MAC1) MML , MAC0 MMOD1	ParaMac16WithMvAndMac16

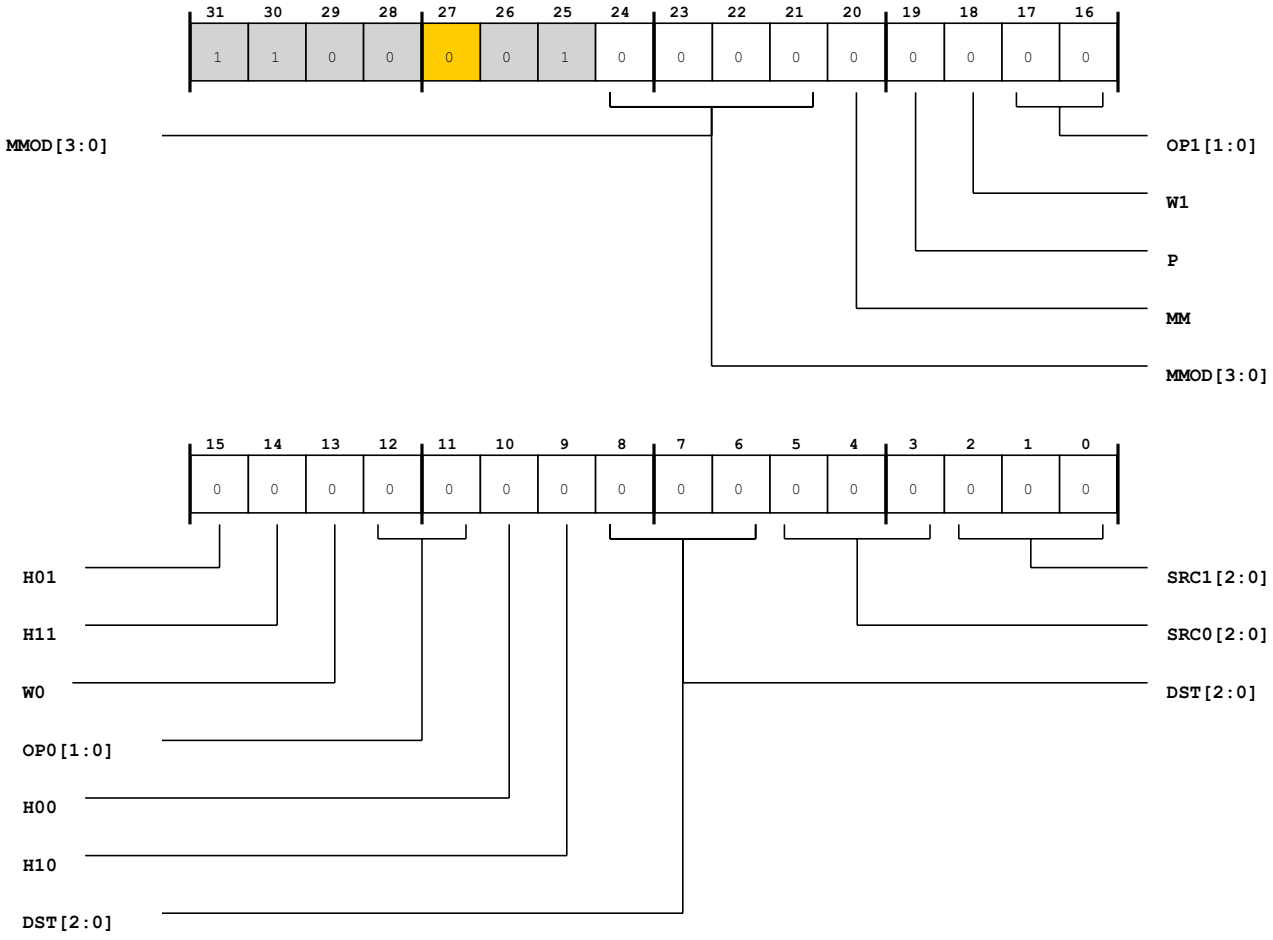
P	W1	W0	OP1	OP0	Syntax	Instruction
0	1	0	10	10	DREG_H =(MAC1) MML , MAC0 MMOD1	ParaMac16WithMvAndMac16
0	1	1	11	11	DREG_H =a1 MML , DREG_L =a0 MMOD1	ParaMvA1ToDregHwithMvA0ToDregL
0	1	1	11	0-	DREG_H =a1 MML , DREG_L =(MAC0) MMOD1	ParaMvAndMac16WithMv
0	1	1	11	10	DREG_H =a1 MML , DREG_L =(MAC0) MMOD1	ParaMvAndMac16WithMv
0	1	1	0-	11	DREG_H =(MAC1) MML , DREG_L =a0 MMOD1	ParaMac16WithMvAndMv
0	1	1	10	11	DREG_H =(MAC1) MML , DREG_L =a0 MMOD1	ParaMac16WithMvAndMv
0	1	1	0-	0-	DREG_H =(MAC1) MML , DREG_L =(MAC0) MMOD1	ParaMac16WithMvAndMac16WithMv
0	1	1	10	0-	DREG_H =(MAC1) MML , DREG_L =(MAC0) MMOD1	ParaMac16WithMvAndMac16WithMv
0	1	1	0-	10	DREG_H =(MAC1) MML , DREG_L =(MAC0) MMOD1	ParaMac16WithMvAndMac16WithMv
0	1	1	10	10	DREG_H =(MAC1) MML , DREG_L =(MAC0) MMOD1	ParaMac16WithMvAndMac16WithMv
1	0	1	11	11	DREG_E =a0 MMODE	MvA0ToDregE
1	0	1	11	0-	DREG_E =(MAC0) MMODE	Mac32WithMv
1	0	1	11	10	DREG_E =(MAC0) MMODE	Mac32WithMv
1	0	1	0-	11	MAC1 MML , DREG_E =a0 MMODE	ParaMac16AndMv
1	0	1	0-	0-	MAC1 MML , DREG_E =(MAC0) MMODE	ParaMac16AndMac16WithMv
1	0	1	0-	10	MAC1 MML , DREG_E =(MAC0) MMODE	ParaMac16AndMac16WithMv
1	0	1	10	11	MAC1 MML , DREG_E =a0 MMODE	ParaMac16AndMv
1	0	1	10	0-	MAC1 MML , DREG_E =(MAC0) MMODE	ParaMac16AndMac16WithMv
1	0	1	10	10	MAC1 MML , DREG_E =(MAC0) MMODE	ParaMac16AndMac16WithMv
1	1	0	11	11	DREG_O =a1 MMLMMODE	MvA1ToDregO
1	1	0	11	0-	DREG_O =a1 MML , MAC0 MMODE	ParaMvAndMac16
1	1	0	11	10	DREG_O =a1 MML , MAC0 MMODE	ParaMvAndMac16
1	1	0	0-	11	DREG_O =(MAC1) MMLMMODE	Mac32WithMv
1	1	0	10	11	DREG_O =(MAC1) MMLMMODE	Mac32WithMv
1	1	0	0-	0-	DREG_O =(MAC1) MML , MAC0 MMODE	ParaMac16WithMvAndMac16

P	W1	W0	OP1	OP0	Syntax	Instruction
1	1	0	0-	10	DREG_O =(MAC1) MML , MACO MMODE	ParaMac16WithMvAndMac16
1	1	0	10	0-	DREG_O =(MAC1) MML , MACO MMODE	ParaMac16WithMvAndMac16
1	1	0	10	10	DREG_O =(MAC1) MML , MACO MMODE	ParaMac16WithMvAndMac16
1	1	1	11	11	DREG_O =a1 MML , DREG_E =a0 MMODE	ParaMvA1ToDregOwithMvA0ToDregE
1	1	1	11	0-	DREG_O =a1 MML , DREG_E =(MACO) MMODE	ParaMvAndMac16WithMv
1	1	1	11	10	DREG_O =a1 MML , DREG_E =(MACO) MMODE	ParaMvAndMac16WithMv
1	1	1	0-	11	DREG_O =(MAC1) MML , DREG_E =a0 MMODE	ParaMac16WithMvAndMv
1	1	1	10	11	DREG_O =(MAC1) MML , DREG_E =a0 MMODE	ParaMac16WithMvAndMv
1	1	1	0-	0-	DREG_O =(MAC1) MML , DREG_E =(MACO) MMODE	ParaMac16WithMvAndMac16WithMv
1	1	1	0-	10	DREG_O =(MAC1) MML , DREG_E =(MACO) MMODE	ParaMac16WithMvAndMac16WithMv
1	1	1	10	0-	DREG_O =(MAC1) MML , DREG_E =(MACO) MMODE	ParaMac16WithMvAndMac16WithMv
1	1	1	10	10	DREG_O =(MAC1) MML , DREG_E =(MACO) MMODE	ParaMac16WithMvAndMac16WithMv

Multiply with 3 operands (Dsp32Mult)

Dsp32Mult Instruction Syntax

Multiply with 3 operands (Dsp32Mult)



The following table provides the opcode field values (OP1, OP0, MM, P, W1, W0), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OP1	OP0	MM	P	W1	W0	Syntax	Instruction	Rev
00	00	0	0	0	1	DREG_L = MULO MMOD1	Mult16	
00	00	-	0	1	0	DREG_H = MUL1 MMLMMOD1	Mult16	
00	00	-	0	1	1	DREG_H = MUL1 MML , DREG_L = MULO MMOD1	ParaMult16AndMult16	
00	00	0	1	0	1	DREG_E = MULO MMODE	Mult16	

OP1	OP0	MM	P	W1	W0	Syntax	Instruction	Rev
00	00	-	1	1	0	DREG_O = MUL1 MMLMMODE	Mult16	
00	00	-	1	1	1	DREG_O = MUL1 MML , DREG_E = MUL0 MMODE	MvAxToDreg	
01	00	0	0	0	0	a1:0 = DREG * DREG M32MMOD	Mac32	2.1
01	01	0	0	0	0	a1:0 += DREG * DREG M32MMOD	Mac32	2.1
01	10	0	0	0	0	a1:0 -= DREG * DREG M32MMOD	Mac32	2.1
01	00	0	0	0	1	DREG = (a1:0 = DREG * DREG) M32MMOD1	Mac32WithMv	2.1
01	01	0	0	0	1	DREG = (a1:0 += DREG * DREG) M32MMOD1	Mac32WithMv	2.1
01	10	0	0	0	1	DREG = (a1:0 -= DREG * DREG) M32MMOD1	Mac32WithMv	2.1
01	11	0	0	0	1	DREG = a1:0 M32MMOD2	MvAxToDreg	2.1
01	00	0	1	0	1	DREG_PAIR = (a1:0 = DREG * DREG) M32MMOD	Mac32WithMv	2.1
01	01	0	1	0	1	DREG_PAIR = (a1:0 += DREG * DREG) M32MMOD	Mac32WithMv	2.1
01	10	0	1	0	1	DREG_PAIR = (a1:0 -= DREG * DREG) M32MMOD	Mac32WithMv	2.1
01	11	0	1	0	1	DREG_PAIR = a1:0 M32MMOD	MvAxToDreg	2.1
01	00	1	0	0	1	DREG = DREG * DREG M32MMOD2	Mult32	2.1
01	00	1	1	0	1	DREG_PAIR = DREG * DREG M32MMOD	Mult32	2.1

M32MMOD

M32MMOD Encode Table

MMOD	Syntax	Description
0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round accumulator 9.31 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MMOD	Syntax	Description
0010	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0011	(is,ns)	The `(is,ns)` option directs the multiplier to
0100	(fu)	The `(fu)` option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1-2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1-2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0110	(iu)	The `(iu)` option directs the multiplier to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum $2^{16}-1$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
0111	(iu,ns)	The `(iu,ns)` option directs the multiplier to
1000	(m)	The `(m)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1010	(m,is)	The `(m,is)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1011	(m,is,ns)	The `(m,is,ns)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format (with no saturation) operation.

M32MMOD1

M32MMOD1 Encode Table

MMOD	Syntax	Description
0001	(t)	The `(t)` option directs the multiplier to use signed fraction with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0010	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0011	(is,ns)	The `(is,ns)` option directs the multiplier to
0101	(tfu)	The `(tfu)` option directs the multiplier to use unsigned fraction with truncation. Multiply 0. 16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1-2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
0110	(iu)	The `(iu)` option directs the multiplier to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum $2^{16}-1$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
0111	(iu,ns)	The `(iu,ns)` option directs the multiplier to
1001	(m,t)	The `(m,t)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with truncation) operation.
1010	(m,is)	The `(m,is)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.

MMOD	Syntax	Description
1011	(m,is,ns)	The `(m,is,ns)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format (with no saturation) operation.

M32MMOD2

M32MMOD2 Encode Table

MMOD	Syntax	Description
0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0001	(t)	The `(t)` option directs the multiplier to use signed fraction with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0010	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0011	(is,ns)	The `(is,ns)` option directs the multiplier to

MMOD	Syntax	Description
0100	(fu)	The `(fu)` option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0101	(tfu)	The `(tfu)` option directs the multiplier to use unsigned fraction with truncation. Multiply 0. 16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
0110	(iu)	The `(iu)` option directs the multiplier to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum $2^{16}\text{-}1$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
0111	(iu,ns)	The `(iu,ns)` option directs the multiplier to
1000	(m)	The `(m)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *Src_reg_0* is the signed operand and *Src_reg_1* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1001	(m,t)	The `(m,t)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with truncation) operation.
1010	(m,is)	The `(m,is)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1011	(m,is,ns)	The `(m,is,ns)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format (with no saturation) operation.

MML

MML Encode Table

MM	Syntax	Description
0		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1	(m)	The `(m)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.

MMLMMOD1

MMLMMOD1 Encode Table

MM	MMOD	Syntax	Description
0	0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MM	MMOD	Syntax	Description
0	0001	(s2rnd)	The (s2rnd) option directs the multiplier to use signed fraction with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	0010	(t)	The (t) option directs the multiplier to use signed fraction with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0	0100	(fu)	The (fu) option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0. 32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0	0110	(tfu)	The (tfu) option directs the multiplier to use unsigned fraction with truncation. Multiply 0. 16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).

MM	MMOD	Syntax	Description
0	1000	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1001	(iss2)	The `(iss2)` option directs the multiplier to use signed integer with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate Accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1011	(ih)	The `(ih)` option directs the multiplier to use signed integer, high word extract. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 32.0 precision; accumulator result is between minimum 0x00 8000 0000 and maximum 0x00 7FFF FFFF. To extract to half-register, round accumulator 40.0 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate to 32.0 result. Copy the upper 16 bits of that value to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0	1100	(iu)	The `(iu)` option directs the multiplier to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum $2^{16}-1$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
1	0000	(m)	The `(m)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *Src_reg_0* is the signed operand and *Src_reg_1* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1	0001	(m,s2rnd)	The `(m,s2rnd)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with scaling and rounding) operation.

MM	MMOD	Syntax	Description
1	0010	(m,t)	The (m,t) option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with truncation) operation.
1	0100	(m,fu)	The (m,fu) option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format operation.
1	0110	(m,tfu)	The (m,tfu) option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format (with truncation) operation.
1	1000	(m,is)	The (m,is) option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1	1001	(m,iss2)	The (m,iss2) option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format with scaling operation.
1	1011	(m,ih)	The (m,ih) option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format (high word extract) operation.
1	1100	(m,iu)	The (m,iu) option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and unsigned integer format operation.

MMLMMODE

MMLMMODE Encode Table

MM	MMOD	Syntax	Description
0	0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round accumulator 9.31 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MM	MMOD	Syntax	Description
0	0001	(s2rnd)	The `(s2rnd)` option directs the multiplier to use signed fraction with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	0100	(fu)	The `(fu)` option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0. 32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0	1000	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16. 0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MM	MMOD	Syntax	Description
0	1001	(iss2)	The `(iss2)` option directs the multiplier to use signed integer with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate Accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1	0000	(m)	The `(m)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *Src_reg_0* is the signed operand and *Src_reg_1* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1	0001	(m,s2rnd)	The `(m,s2rnd)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with scaling and rounding) operation.
1	0100	(m,fu)	The `(m,fu)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format operation.
1	1000	(m,is)	The `(m,is)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1	1001	(m,iss2)	The `(m,iss2)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format with scaling operation.

MMOD1

MMOD1 Encode Table

MMOD	Syntax	Description
0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0001	(s2rnd)	The `(s2rnd)` option directs the multiplier to use signed fraction with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0010	(t)	The `(t)` option directs the multiplier to use signed fraction with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).

MMOD	Syntax	Description
0100	(fu)	The `(fu)` option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1-2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1-2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0110	(tfu)	The `(tfu)` option directs the multiplier to use unsigned fraction with truncation. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1-2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
1000	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1001	(iss2)	The `(iss2)` option directs the multiplier to use signed integer with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate Accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MMOD	Syntax	Description
1011	(ih)	The (ih) option directs the multiplier to use signed integer, high word extract. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 32.0 precision; accumulator result is between minimum 0x00 8000 0000 and maximum 0x00 7FFF FFFF. To extract to half-register, round accumulator 40.0 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate to 32.0 result. Copy the upper 16 bits of that value to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
1100	(iu)	The (iu) option directs the multiplier to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum $2^{16}-1$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).

MMODE

MMODE Encode Table

MMOD	Syntax	Description
0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, round accumulator 9.31 format value at bit 16. (The ASTAT.RND_MOD bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MMOD	Syntax	Description
0001	(s2rnd)	The `(s2rnd)` option directs the multiplier to use signed fraction with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0100	(fu)	The `(fu)` option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To **extract to half register**, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1-2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To **extract to full register**, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1-2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
1000	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1001	(iss2)	The `(iss2)` option directs the multiplier to use signed integer with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate Accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To **extract to half register**, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To **extract to full register**, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MUL0

MUL0 Encode Table

H00	H10	Syntax
0	0	DREG_L * DREG_L
0	1	DREG_L * DREG_H
1	0	DREG_H * DREG_L
1	1	DREG_H * DREG_H

MUL1

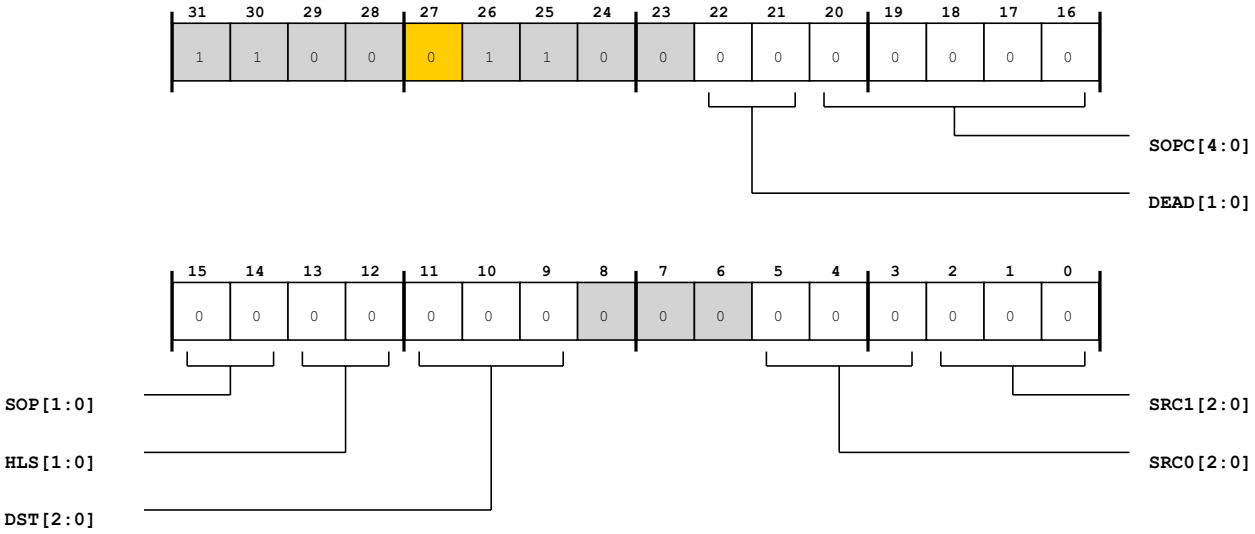
MUL1 Encode Table

H01	H11	Syntax
0	0	DREG_L * DREG_L
0	1	DREG_L * DREG_H
1	0	DREG_H * DREG_L
1	1	DREG_H * DREG_H

Shift (Dsp32Shf)

Dsp32Shf Instruction Syntax

Shift (Dsp32Shf)



The following table provides the opcode field values (SOPC, SOP, HLS), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

SOPC	SOP	HLS	Syntax	Instruction
00000	00	00	DREG_L = ashift DREG_L by DREG_L	AShift16
00000	00	01	DREG_L = ashift DREG_H by DREG_L	AShift16
00000	00	10	DREG_H = ashift DREG_L by DREG_L	AShift16
00000	00	11	DREG_H = ashift DREG_H by DREG_L	AShift16
00000	01	00	DREG_L = ashift DREG_L by DREG_L (s)	AShift16
00000	01	01	DREG_L = ashift DREG_H by DREG_L (s)	AShift16
00000	01	10	DREG_H = ashift DREG_L by DREG_L (s)	AShift16
00000	01	11	DREG_H = ashift DREG_H by DREG_L (s)	AShift16
00000	10	00	DREG_L = lshift DREG_L by DREG_L	LShift16
00000	10	01	DREG_L = lshift DREG_H by DREG_L	LShift16
00000	10	10	DREG_H = lshift DREG_L by DREG_L	LShift16
00000	10	11	DREG_H = lshift DREG_H by DREG_L	LShift16
00001	00	00	DREG = ashift DREG by DREG_L (v)	AShift16Vec
00001	01	00	DREG = ashift DREG by DREG_L (v,s)	AShift16Vec

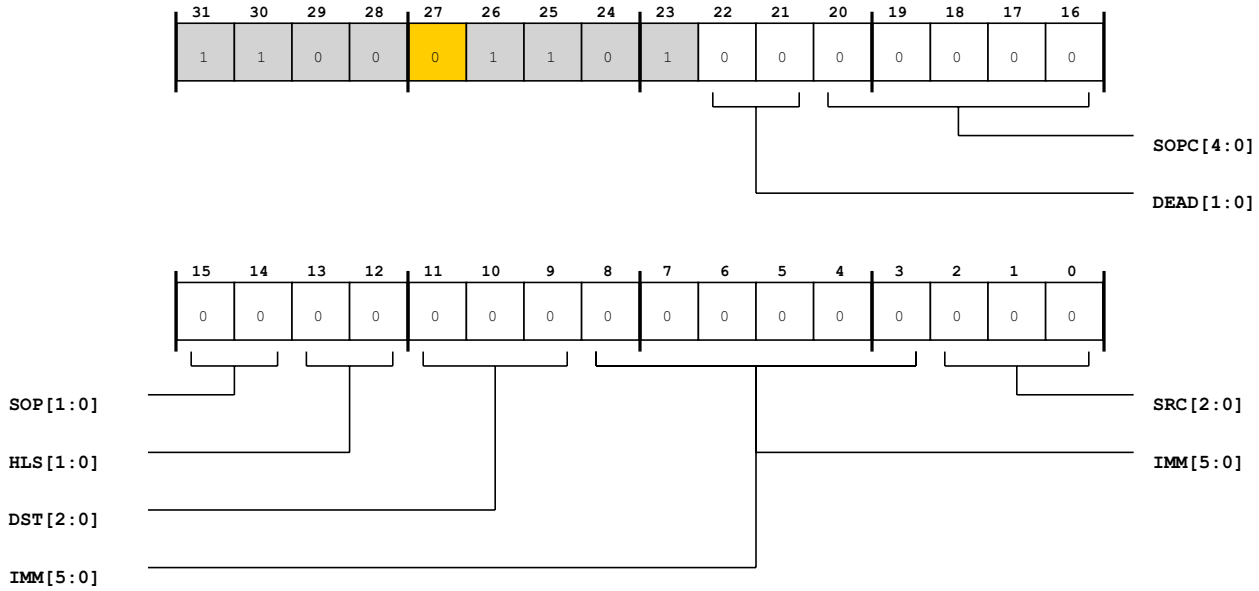
SOPC	SOP	HLS	Syntax	Instruction
00001	10	00	DREG = lshift DREG by DREG_L (v)	LShift16Vec
00010	00	00	DREG = ashift DREG by DREG_L	AShift32
00010	01	00	DREG = ashift DREG by DREG_L (s)	AShift32
00010	10	00	DREG = lshift DREG by DREG_L	LShift
00010	11	00	DREG = rot DREG by DREG_L	Shift_Rot32
00011	00	00	a0 = ashift a0 by DREG_L	AShiftAcc
00011	00	01	a1 = ashift a1 by DREG_L	AShiftAcc
00011	01	00	a0 = lshift a0 by DREG_L	LShiftA
00011	01	01	a1 = lshift a1 by DREG_L	LShiftA
00011	10	00	a0 = rot a0 by DREG_L	Shift_RotAcc
00011	10	01	a1 = rot a1 by DREG_L	Shift_RotAcc
00100	00	00	DREG = pack (DREG_L , DREG_L)	Pack16Vec
00100	01	00	DREG = pack (DREG_L , DREG_H)	Pack16Vec
00100	10	00	DREG = pack (DREG_H , DREG_L)	Pack16Vec
00100	11	00	DREG = pack (DREG_H , DREG_H)	Pack16Vec
00101	00	00	DREG_L = signbits DREG	Shift_SignBits32
00101	01	00	DREG_L = signbits DREG_L	Shift_SignBits32
00101	10	00	DREG_L = signbits DREG_H	Shift_SignBits32
00110	00	00	DREG_L = signbits a0	Shift_SignBitsAcc
00110	01	00	DREG_L = signbits a1	Shift_SignBitsAcc
00110	11	00	DREG_L = ones DREG	Shift_Ones
00111	00	00	DREG_L = expadj (DREG , DREG_L)	Shift_ExpAdj32
00111	01	00	DREG_L = expadj (DREG , DREG_L) (v)	Shift_ExpAdj32
00111	10	00	DREG_L = expadj (DREG_L , DREG_L)	Shift_ExpAdj32
00111	11	00	DREG_L = expadj (DREG_H , DREG_L)	Shift_ExpAdj32
01000	00	00	bitmux (DREG , DREG , a0) (asr)	BitMux
01000	01	00	bitmux (DREG , DREG , a0) (asl)	BitMux
01001	00	00	DREG_L = vit_max (DREG) (asl)	Shift_VitMax
01001	01	00	DREG_L = vit_max (DREG) (asr)	Shift_VitMax
01001	10	00	DREG = vit_max (DREG , DREG) (asl)	Shift_DualVitMax
01001	11	00	DREG = vit_max (DREG , DREG) (asr)	Shift_DualVitMax
01010	00	00	DREG = extract (DREG , DREG_L) (z)	Shift_Extract
01010	01	00	DREG = extract (DREG , DREG_L) (x)	Shift_Extract
01010	10	00	DREG = deposit (DREG , DREG)	Shift_Deposit
01010	11	00	DREG = deposit (DREG , DREG) (x)	Shift_Deposit
01011	00	00	DREG_L = cc = bxorshift (a0, DREG)	BXOR_NF

SOPC	SOP	HLS	Syntax	Instruction
01011	01	00	DREG_L = cc = bxor (a0, DREG)	BXOR_NF
01100	00	00	a0 = bxorshift (a0, a1, cc)	BXORShift_NF
01100	01	00	DREG_L = cc = bxor (a0, a1, cc)	BXOR
01101	00	00	DREG = align8 (DREG , DREG)	Shift_Align
01101	01	00	DREG = align16 (DREG , DREG)	Shift_Align
01101	10	00	DREG = align24 (DREG , DREG)	Shift_Align

Shift Immediate (Dsp32Shfimm)

Dsp32Shfimm Instruction Syntax

Shift Immediate (Dsp32Shfimm)



The following table provides the opcode field values (SOPC, SOP, HLS), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

SOPC	SOP	HLS	Syntax	Instruction
00000	00	00	DREG_L = DREG_L AHSH4	AShift16
00000	00	01	DREG_L = DREG_H AHSH4	AShift16
00000	00	10	DREG_H = DREG_L AHSH4	AShift16
00000	00	11	DREG_H = DREG_H AHSH4	AShift16
00000	01	00	DREG_L = DREG_L AHSH4S	AShift16
00000	01	01	DREG_L = DREG_H AHSH4S	AShift16
00000	01	10	DREG_H = DREG_L AHSH4S	AShift16
00000	01	11	DREG_H = DREG_H AHSH4S	AShift16
00000	10	00	DREG_L = DREG_L LHSH4	LShift16
00000	10	01	DREG_L = DREG_H LHSH4	LShift16
00000	10	10	DREG_H = DREG_L LHSH4	LShift16
00000	10	11	DREG_H = DREG_H LHSH4	LShift16
00001	00	00	DREG = DREG AHSH4 (v)	AShift16Vec

SOPC	SOP	HLS	Syntax	Instruction
00001	01	00	DREG = DREG AHSH4VS	AShift16Vec
00001	10	00	DREG = DREG LHSH4 (v)	LShift16Vec
00010	00	00	DREG = DREG ASH5	AShift32
00010	01	00	DREG = DREG ASH5S	AShift32
00010	10	00	DREG = DREG LSH5	LShift
00010	11	00	DREG = rot DREG by imm6	Shift_Rot32
00011	00	00	a0 = a0 ASH5	AShiftAcc
00011	00	01	a1 = a1 ASH5	AShiftAcc
00011	01	00	a0 = a0 LSH5	LShiftA
00011	01	01	a1 = a1 LSH5	LShiftA
00011	10	00	a0 = rot a0 by imm6	Shift_RotAcc
00011	10	01	a1 = rot a1 by imm6	Shift_RotAcc

AHSH4

AHSH4 Encode Table

IMM	Syntax	Rev
000000	<<< 0	2.1.1
00----	<<< uimm4nz	2.1.1
11----	>>> uimm4nznegpos	

AHSH4S

AHSH4S Encode Table

IMM	Syntax	Rev
000000	<< 0 (s)	
00----	<< uimm4nz (s)	
11----	>>> uimm4nznegpos (s)	2.1.1

AHSH4VS

AHSH4VS Encode Table

IMM	Syntax	Rev
000000	<< 0 (v,s)	
00----	<< uimm4nz (v,s)	
11----	>>> uimm4nznegpos (v,s)	2.1.1

ASH5

ASH5 Encode Table

IMM	Syntax	Rev
000000	<<< 0	2.1.1
0-----	<<< uimm5nz	2.1.1
1-----	>>> uimm5nznegpos	

ASH5S

ASH5S Encode Table

IMM	Syntax	Rev
000000	<< 0 (s)	
0-----	<< uimm5nz (s)	
1-----	>>> uimm5nznegpos (s)	2.1.1

LHSH4

LHSH4 Encode Table

IMM	Syntax
000000	<< 0

IMM	Syntax
00----	<< uimm4nz
11----	>> uimm4nznegpos

LSH5

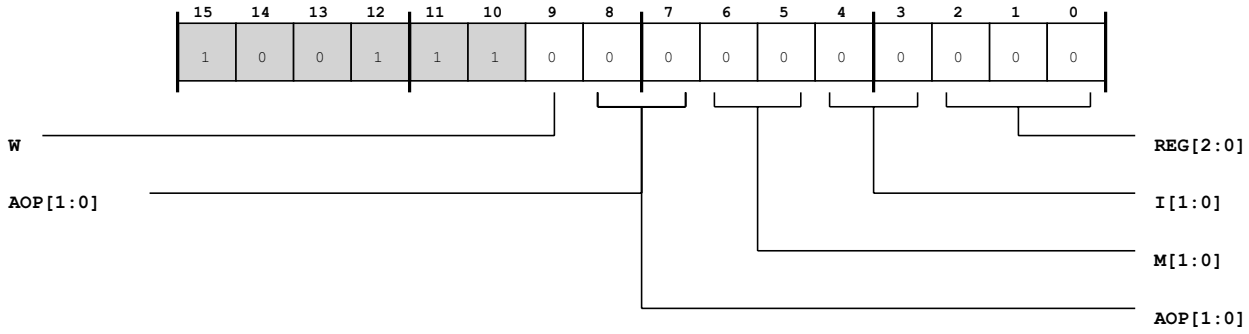
LSH5 Encode Table

IMM	Syntax
000000	<< 0
0-----	<< uimm5nz
1-----	>> uimm5nznegpos

Load/Store (DspLdSt)

DspLdSt Instruction Syntax

Load/Store (DspLdSt)



The following table provides the opcode field values (W, M, AOP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

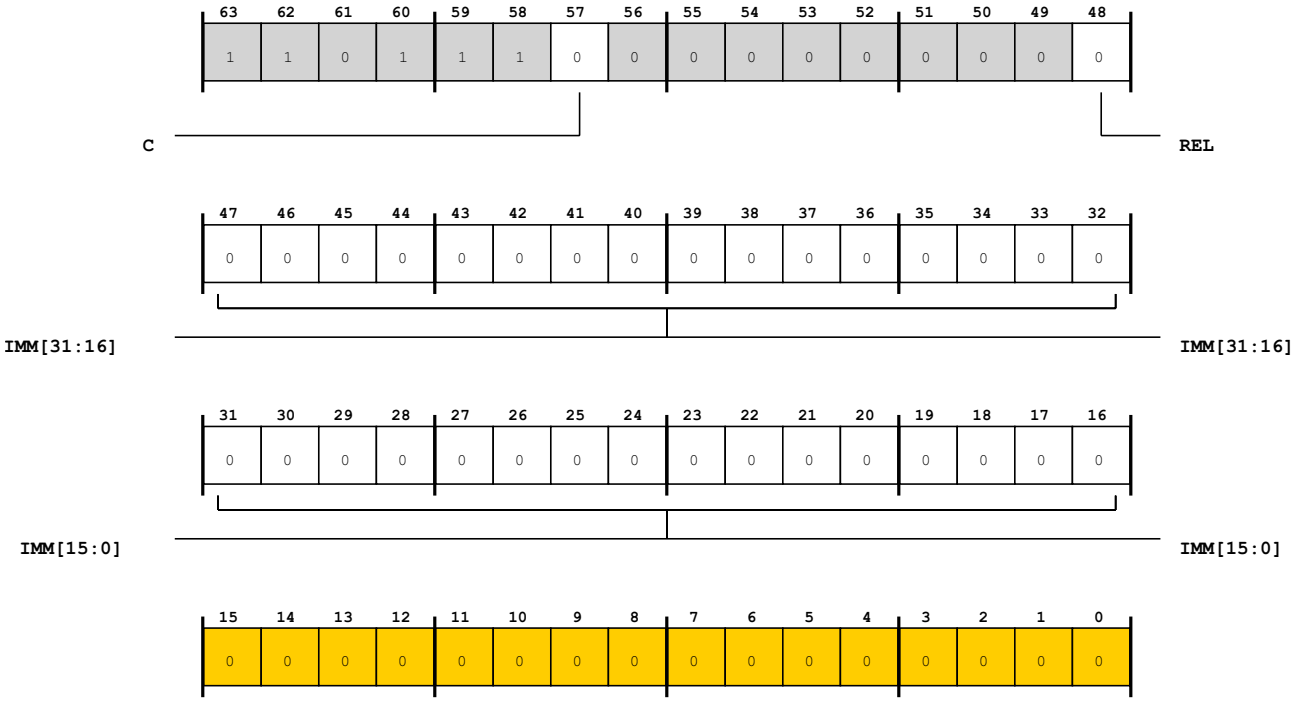
W	M	AOP	Syntax	Instruction
0	00	00	DREG =[IREG ++]	LdM32bitToDreg
0	00	01	DREG =[IREG --]	LdM32bitToDreg
0	00	10	DREG =[IREG]	LdM32bitToDreg
0	01	00	DREG_L =w[IREG ++]	LdM16bitToDregL
0	01	01	DREG_L =w[IREG --]	LdM16bitToDregL
0	01	10	DREG_L =w[IREG]	LdM16bitToDregL
0	10	00	DREG_H =w[IREG ++]	LdM16bitToDregH
0	10	01	DREG_H =w[IREG --]	LdM16bitToDregH
0	10	10	DREG_H =w[IREG]	LdM16bitToDregH
0	--	11	DREG =[IREG ++ MREG]	LdM32bitToDreg
1	00	00	[IREG ++]= DREG	StDregToM32bit
1	00	01	[IREG --]= DREG	StDregToM32bit
1	00	10	[IREG]= DREG	StDregToM32bit
1	01	00	w[IREG ++]= DREG_L	StDregLToM16bit
1	01	01	w[IREG --]= DREG_L	StDregLToM16bit
1	01	10	w[IREG]= DREG_L	StDregLToM16bit
1	10	00	w[IREG ++]= DREG_H	StDregHToM16bit
1	10	01	w[IREG --]= DREG_H	StDregHToM16bit

W	M	AOP	Syntax	Instruction
1	10	10	w[IREG] = DREG_H	StDregHToM16bit
1	--	11	[IREG ++ MREG] = DREG	StDregToM32bit

Jump/Call to 32-bit Immediate (Jump32)

Jump32 Instruction Syntax

Jump/Call to 32-bit Immediate (Jump32)



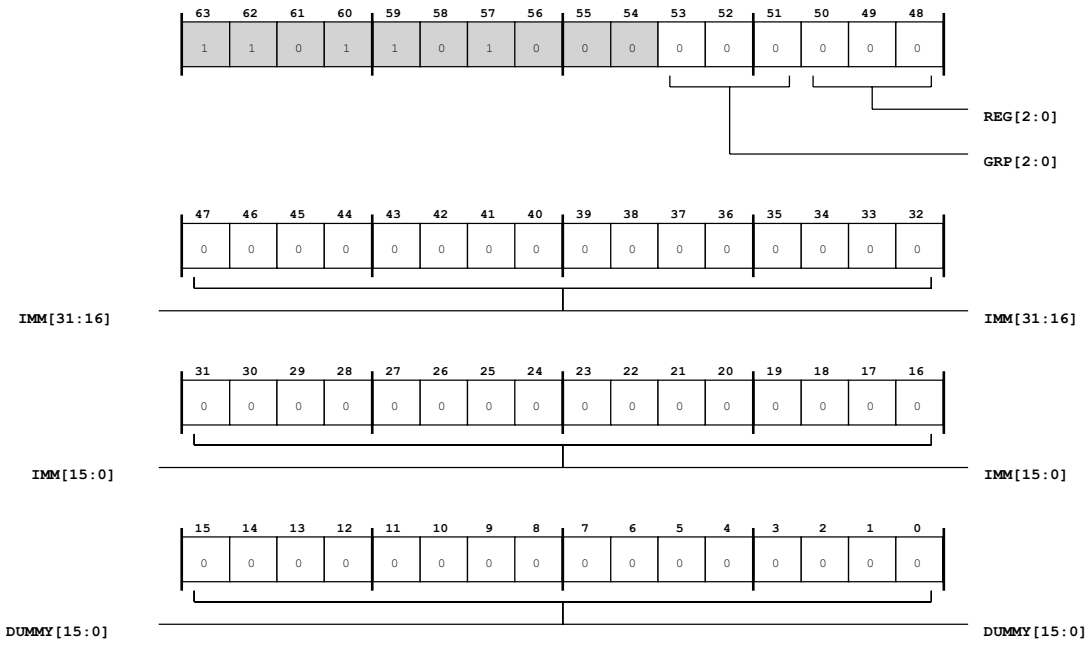
The following table provides the opcode field values (C, REL), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

C	REL	Syntax	Instruction
0	0	jump.a buimm32	JumpAbs
0	1	jump bimm32	JumpAbs
1	0	call.a buimm32	Call
1	1	call bimm32	Call

Load Immediate Word (LdImm)

LdImm Instruction Syntax

Load Immediate Word (LdImm)



The following table provides the opcode field values (GRP, REG), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

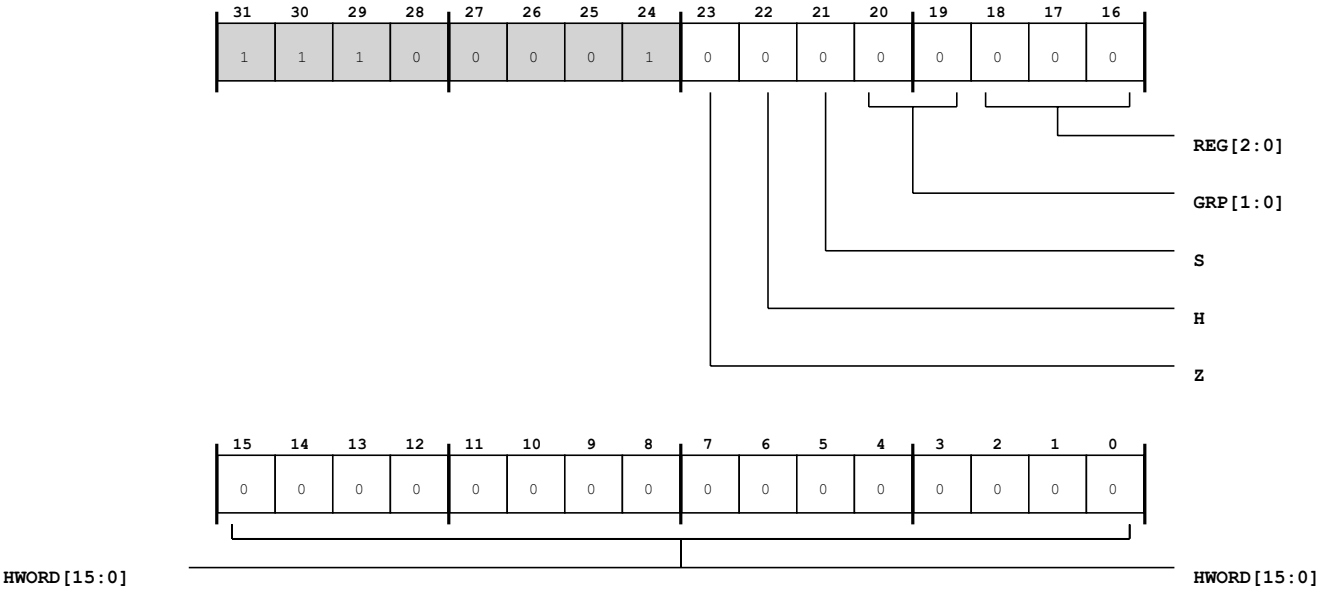
GRP	REG	Syntax	Instruction
000	---	DREG = imm32	LdImmToReg
001	---	PREG = imm32	LdImmToReg
010	0--	IREG = imm32	LdImmToReg
010	1--	MREG = imm32	LdImmToReg
011	0--	BREG = imm32	LdImmToReg
011	1--	LREG = imm32	LdImmToReg
100	000	a0.x = imm32	LdImmToAxX
100	001	a0.w = imm32	LdImmToAxW
100	010	a1.x = imm32	LdImmToAxX
100	011	a1.w = imm32	LdImmToAxW
100	110	astat = imm32	LdImmToReg
100	111	rets = imm32	LdImmToReg

GRP	REG	Syntax	Instruction
110	---	SYSREG2 = imm32	LdImmToReg
111	---	SYSREG3 = imm32	LdImmToReg

Load Immediate Half Word (LdImmHalf)

LdImmHalf Instruction Syntax

Load Immediate Half Word (LdImmHalf)



The following table provides the opcode field values (H, Z, S), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

H	Z	S	Syntax	Instruction
0	0	0	DST_L = imm16	LdImmToDregHL
0	0	1	DST = imm16 (x)	LdImmToReg
0	1	0	DST = rimm16 (z)	LdImmToReg
1	0	0	DST_H = imm16	LdImmToDregHL

DST

DST Encode Table

GRP	REG	Syntax
00	---	DREG
01	---	PREG

GRP	REG	Syntax
10	0--	IREG
10	1--	MREG
11	0--	BREG
11	1--	LREG

DST_H

DST_H Encode Table

GRP	REG	Syntax
00	---	DREG_H
01	---	PREG_H
10	0--	IREG_H
10	1--	MREG_H
11	0--	BREG_H
11	1--	LREG_H

DST_L

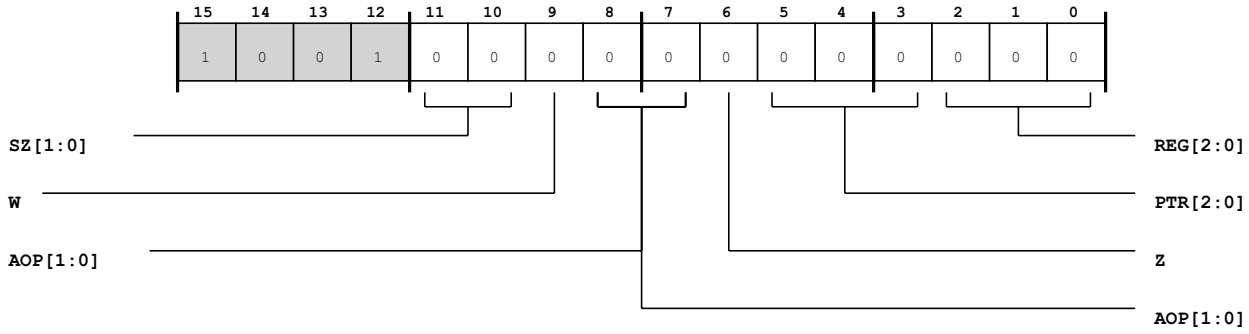
DST_L Encode Table

GRP	REG	Syntax
00	---	DREG_L
01	---	PREG_L
10	0--	IREG_L
10	1--	MREG_L
11	0--	BREG_L
11	1--	LREG_L

Load/Store (LdSt)

LdSt Instruction Syntax

Load/Store (LdSt)



The following table provides the opcode field values (W, SZ, Z, AOP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

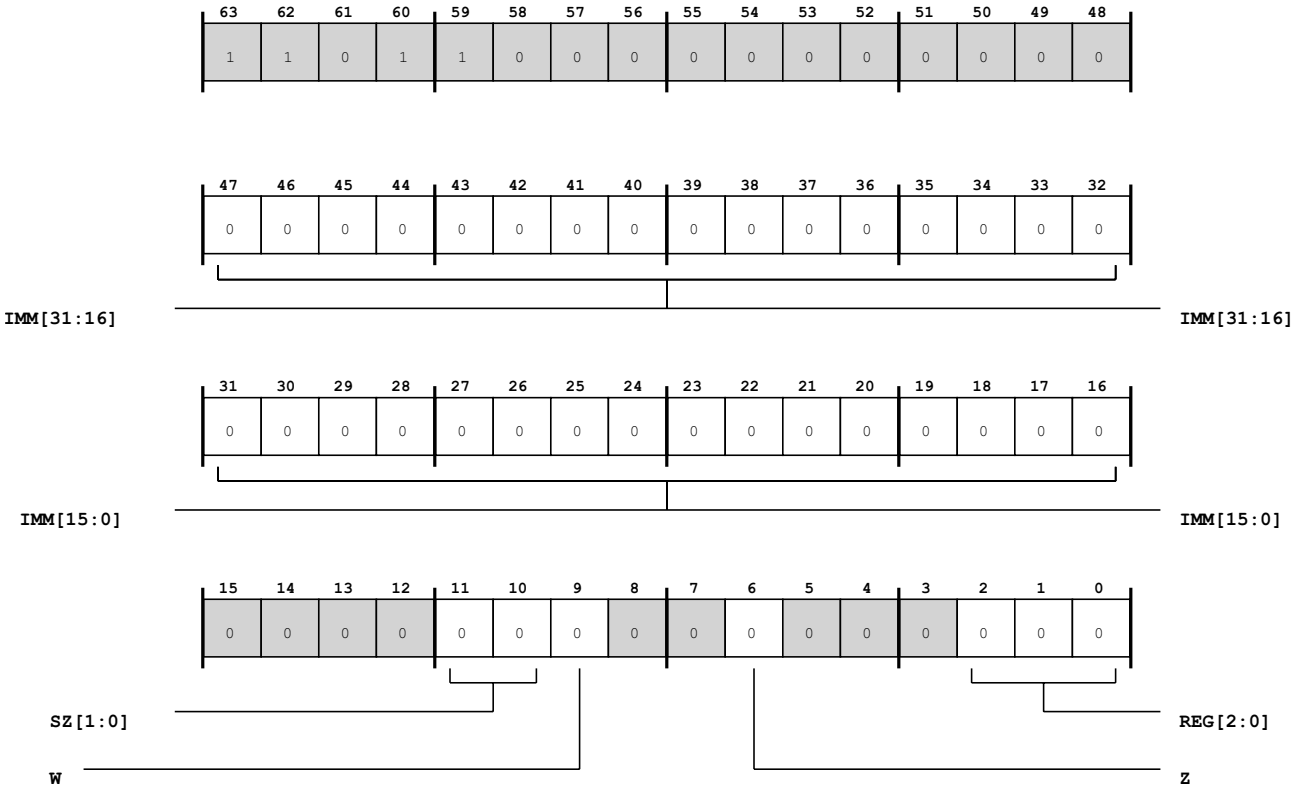
W	SZ	Z	AOP	Syntax	Instruction
0	00	0	00	DREG = [PREG ++]	LdM32bitToDreg
0	00	0	01	DREG = [PREG --]	LdM32bitToDreg
0	00	0	10	DREG = [PREG]	LdM32bitToDreg
0	01	0	00	DREG = w[PREG ++] (z)	LdM16bitToDreg
0	01	0	01	DREG = w[PREG --] (z)	LdM16bitToDreg
0	01	0	10	DREG = w[PREG] (z)	LdM16bitToDreg
0	01	1	00	DREG = w[PREG ++] (x)	LdM16bitToDreg
0	01	1	01	DREG = w[PREG --] (x)	LdM16bitToDreg
0	01	1	10	DREG = w[PREG] (x)	LdM16bitToDreg
0	10	0	00	DREG = b[PREG ++] (z)	LdM08bitToDreg
0	10	0	01	DREG = b[PREG --] (z)	LdM08bitToDreg
0	10	0	10	DREG = b[PREG] (z)	LdM08bitToDreg
0	10	1	00	DREG = b[PREG ++] (x)	LdM08bitToDreg
0	10	1	01	DREG = b[PREG --] (x)	LdM08bitToDreg
0	10	1	10	DREG = b[PREG] (x)	LdM08bitToDreg
1	00	0	00	[PREG ++] = DREG	StDregToM32bit
1	00	0	01	[PREG --] = DREG	StDregToM32bit
1	00	0	10	[PREG] = DREG	StDregToM32bit

W	SZ	Z	AOP	Syntax	Instruction
1	00	1	00	[PREG ++] = PREG	StPregToM32bit
1	00	1	01	[PREG --] = PREG	StPregToM32bit
1	00	1	10	[PREG] = PREG	StPregToM32bit
1	01	0	00	w[PREG ++] = DREG	StDregLToM16bit
1	01	0	01	w[PREG --] = DREG	StDregLToM16bit
1	01	0	10	w[PREG] = DREG	StDregLToM16bit
1	10	0	00	b[PREG ++] = DREG	StDregToM08bit
1	10	0	01	b[PREG --] = DREG	StDregToM08bit
1	10	0	10	b[PREG] = DREG	StDregToM08bit

Load/Store 32-bit Absolute Address (LdStAbs)

LdStAbs Instruction Syntax

Load/Store 32-bit Absolute Address (LdStAbs)



The following table provides the opcode field values (W, SZ, Z), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

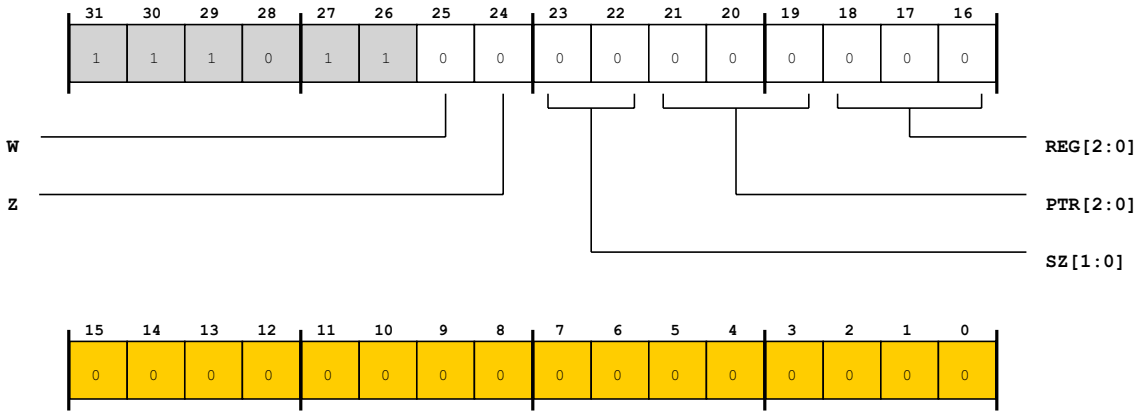
W	SZ	Z	Syntax	Instruction
0	00	0	DREG = [uimm32]	LdM32bitToDreg
0	00	1	PREG = [uimm32]	LdM32bitToDreg
0	01	0	DREG = w[uimm32] (z)	LdM16bitToDreg
0	01	1	DREG = w[uimm32] (x)	LdM16bitToDreg
0	10	0	DREG = b[uimm32] (z)	LdM08bitToDreg
0	10	1	DREG = b[uimm32] (x)	LdM08bitToDreg
0	11	0	DREG_L = w[uimm32]	LdM16bitToDregL
0	11	1	DREG_H = w[uimm32]	LdM16bitToDregH

W	SZ	Z	Syntax	Instruction
1	00	0	[uimm32] = DREG	StDregToM32bit
1	00	1	[uimm32] = PREG	StDregToM32bit
1	01	0	w[uimm32] = DREG	StDregLToM16bit
1	10	0	b[uimm32] = DREG	StDregToM08bit
1	11	1	w[uimm32] = DREG_H	StDregHToM16bit

Long Load/Store with indexed addressing (LdSt-Excl)

LdStExcl Instruction Syntax

Long Load/Store with indexed addressing (LdStExcl)



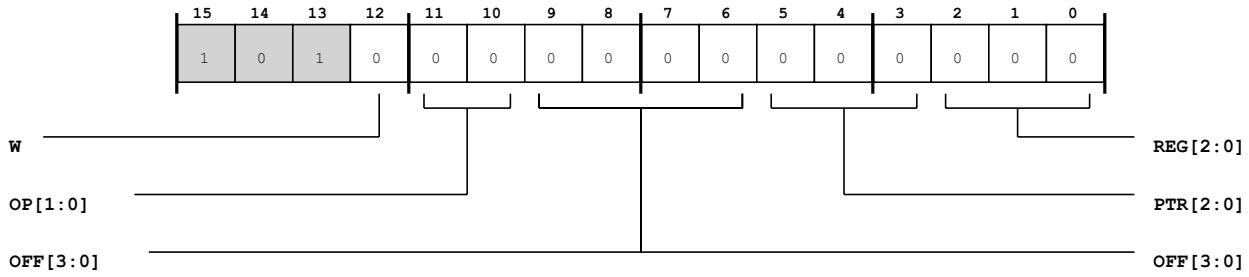
The following table provides the opcode field values (W, SZ, Z), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	SZ	Z	Syntax	Instruction	Rev
0	00	0	DREG = [PREG] (excl)	LdX32bitToDreg	2.2
0	01	0	DREG = w[PREG] (z,excl)	LdX16bitToDreg	2.2
0	01	1	DREG = w[PREG] (x,excl)	LdX16bitToDreg	2.2
0	10	0	DREG = b[PREG] (z,excl)	LdX08bitToDreg	2.2
0	10	1	DREG = b[PREG] (x,excl)	LdX08bitToDreg	2.2
0	11	0	DREG_L = w[PREG] (excl)	LdX16bitToDregL	2.2
0	11	1	DREG_H = w[PREG] (excl)	LdX16bitToDregH	2.2
1	00	0	cc = ([PREG] = DREG) (excl)	StDregToX32bit	2.2
1	01	0	cc = (w[PREG] = DREG) (excl)	StDregLToX16bit	2.2
1	10	0	cc = (b[PREG] = DREG) (excl)	StDregToX08bit	2.2
1	11	0	cc = (w[PREG] = DREG_H) (excl)	StDregHToX16bit	2.2
1	11	1	syncexcl	SyncExcl	2.2

Load/Store indexed with small immediate offset (LdStII)

LdStII Instruction Syntax

Load/Store indexed with small immediate offset (LdStII)



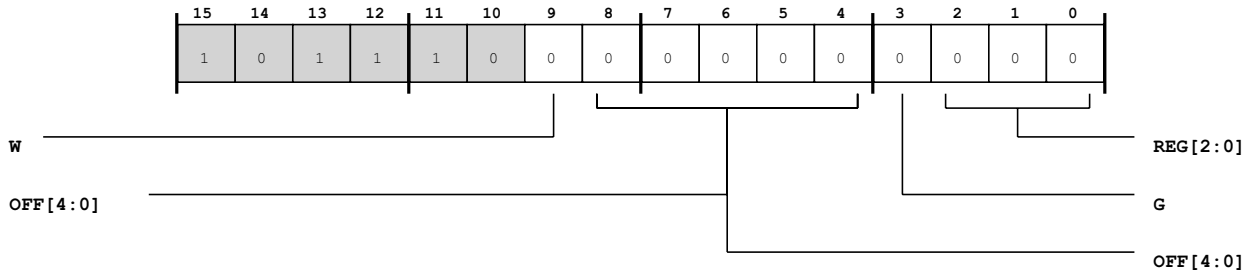
The following table provides the opcode field values (W, OP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	OP	Syntax	Instruction
0	00	DREG = [PREG + uimm4s4]	LdM32bitToDreg
0	01	DREG = w[PREG + uimm4s2] (z)	LdM16bitToDreg
0	10	DREG = w[PREG + uimm4s2] (x)	LdM16bitToDreg
1	00	[PREG + uimm4s4] = DREG	StDregToM32bit
1	01	w[PREG + uimm4s2] = DREG	StDregLToM16bit
1	11	[PREG + uimm4s4] = PREG	StPregToM32bit

Load/Store indexed with small immediate offset FP (LdStIIFP)

LdStIIFP Instruction Syntax

Load/Store indexed with small immediate offset FP (LdStIIFP)



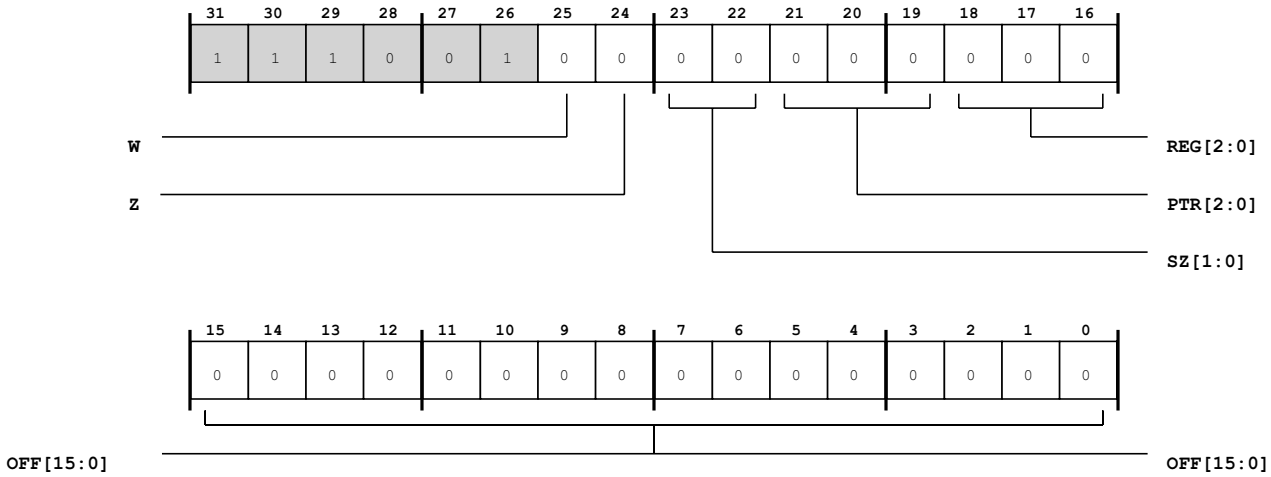
The following table provides the opcode field values (W, G), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	G	Syntax	Instruction
0	0	DREG = [fp - imm5nzs4negpos]	LdM32bitToDreg
1	0	[fp - imm5nzs4negpos] = DREG	StDregToM32bit
1	1	[fp - imm5nzs4negpos] = PREG	StDregToM32bit

Long Load/Store with indexed addressing (LdStIdxI)

LdStIdxI Instruction Syntax

Long Load/Store with indexed addressing (LdStIdxI)



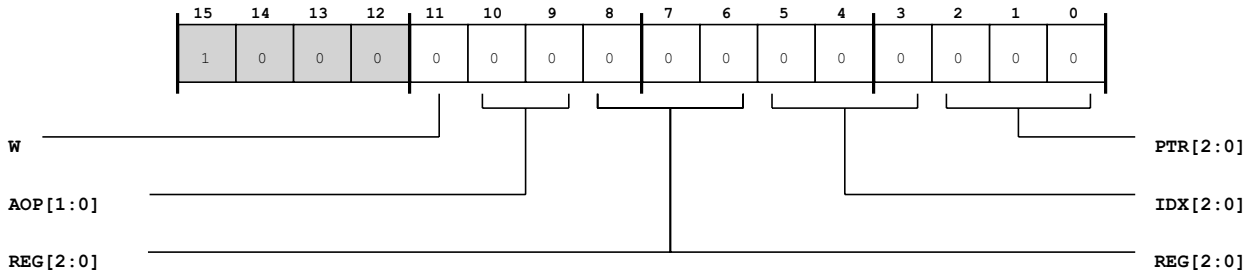
The following table provides the opcode field values (W, SZ, Z), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	SZ	Z	Syntax	Instruction
0	00	0	DREG = [PREG + imm16s4]	LdM32bitToDreg
0	00	1	PREG = [PREG + imm16s4]	LdM32bitToPreg
0	01	0	DREG = w[PREG + imm16s2] (z)	LdM16bitToDreg
0	01	1	DREG = w[PREG + imm16s2] (x)	LdM16bitToDreg
0	10	0	DREG = b[PREG + imm16reloc] (z)	LdM08bitToDreg
0	10	1	DREG = b[PREG + imm16reloc] (x)	LdM08bitToDreg
1	00	0	[PREG + imm16s4] = DREG	StDregToM32bit
1	00	1	[PREG + imm16s4] = PREG	StPregToM32bit
1	01	0	w[PREG + imm16s2] = DREG	StDregLToM16bit
1	10	0	b[PREG + imm16reloc] = DREG	StDregToM08bit

Load/Store post modify addressing, p-register based (LdStPmod)

LdStPmod Instruction Syntax

Load/Store postmodify addressing, pregister based (LdStPmod)



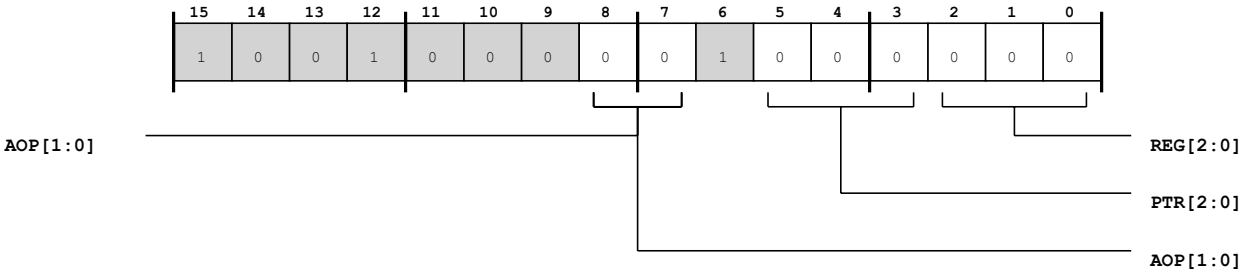
The following table provides the opcode field values (W, AOP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	AOP	Syntax	Instruction
0	00	DREG = [PREG ++ PREG]	LdM32bitToDreg
0	01	DREG_L = w[PREG ++ PREG]	LdM16bitToDregL
0	10	DREG_H = w[PREG ++ PREG]	LdM16bitToDregH
0	11	DREG = w[PREG ++ PREG] (z)	LdM16bitToDreg
1	00	[PREG ++ PREG] = DREG	StDregToM32bit
1	01	w[PREG ++ PREG] = DREG_L	StDregLToM16bit
1	10	w[PREG ++ PREG] = DREG_H	StDregHToM16bit
1	11	DREG = w[PREG ++ PREG] (x)	LdM16bitToDreg

Load/Store (Ldp)

Ldp Instruction Syntax

Load/Store (Ldp)



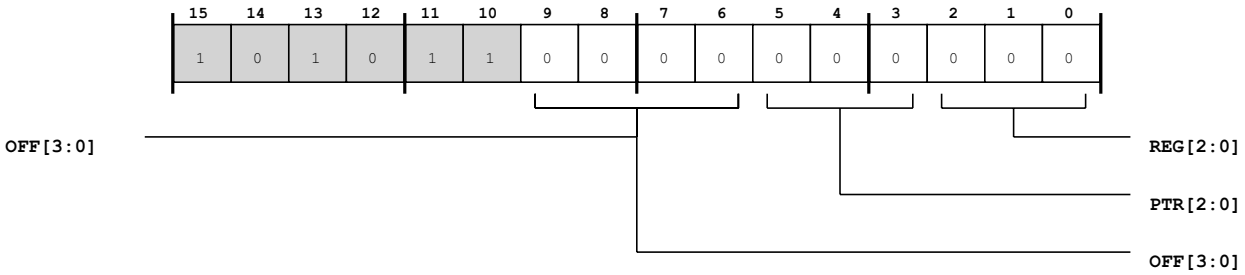
The following table provides the opcode field values (AOP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

AOP	Syntax	Instruction
00	PREG =[PREG ++]	LdM32bitToPreg
01	PREG =[PREG --]	LdM32bitToPreg
10	PREG =[PREG]	LdM32bitToPreg

Load/Store indexed with small immediate offset (LdpII)

LdpII Instruction Syntax

Load/Store indexed with small immediate offset (LdpII)



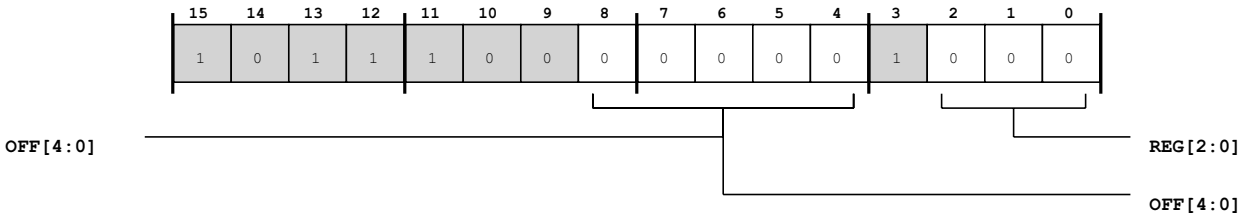
The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

	Syntax	Instruction
	PREG =[PREG + uimm4s4]	LdM32bitToPreg

Load/Store indexed with small immediate offset FP (LdpIIFP)

LdpIIFP Instruction Syntax

Load/Store indexed with small immediate offset FP (LdpIIFP)



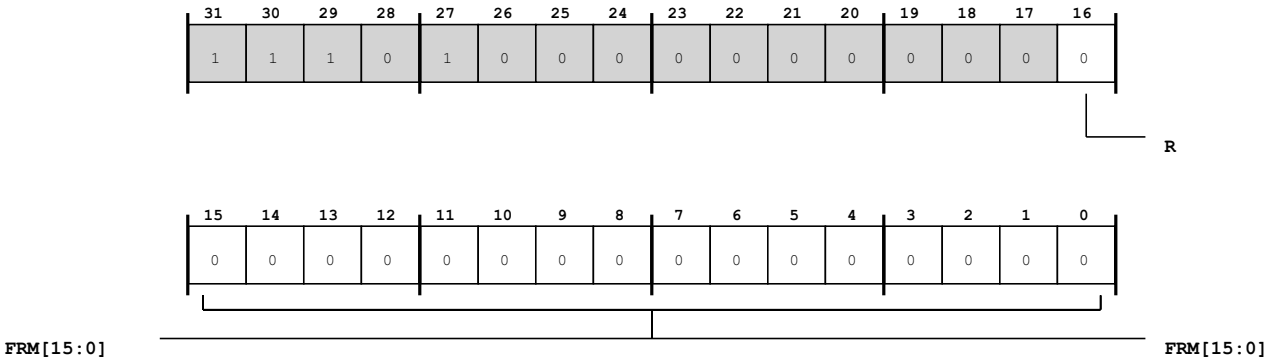
The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

	Syntax	Instruction
	PREG = [fp - imm5nzs4negpos]	LdM32bitToDreg

Save/restore registers and link/unlink frame, multiple cycles (Linkage)

Linkage Instruction Syntax

Save/restore registers and link/unlink frame, multiple cycles (Linkage)



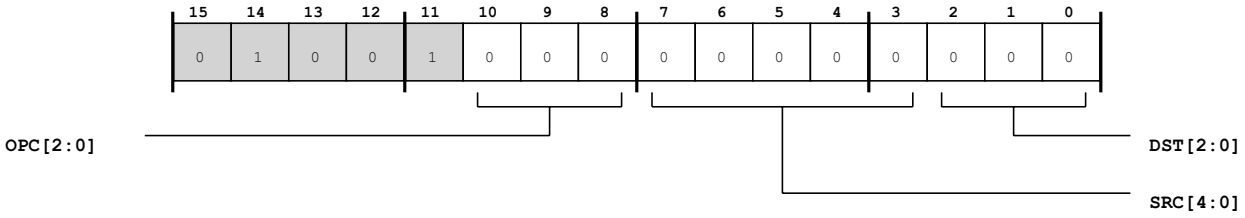
The following table provides the opcode field values (R), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

R	Syntax	Instruction
0	link uimm16s4	Linkage
1	unlink	Linkage

Logic Binary Operations (Logi2Op)

Logi2Op Instruction Syntax

Logic Binary Operations (Logi2Op)



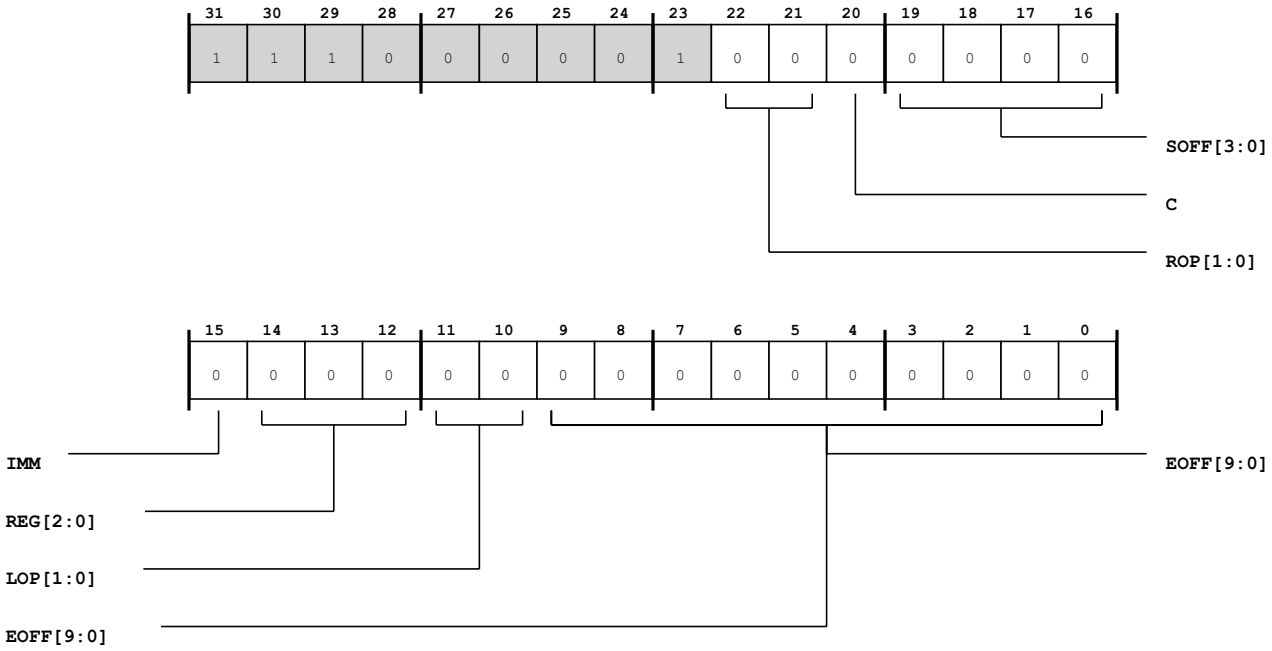
The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
000	cc = !bittst (DREG , uimm5)	Shift_BitTst
001	cc = bittst (DREG , uimm5)	Shift_BitTst
010	bitset (DREG , uimm5)	Shift_BitMod
011	bittgl (DREG , uimm5)	Shift_BitMod
100	bitclr (DREG , uimm5)	Shift_BitMod
101	DREG >>>= uimm5	AShift32
110	DREG >>= uimm5	LShift
111	DREG <<= uimm5	LShift

Virtually Zero Overhead Loop Mechanism (Loop-Setup)

LoopSetup Instruction Syntax

Virtually Zero Overhead Loop Mechanism (LoopSetup)



The following table provides the opcode field values (LOP, ROP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

LOP	ROP	Syntax	Instruction	Rev
00	00	lsetup (uimm4s2o4 , uimm10s2o4) LC	LoopSetup	
00	01	lsetup (uimm4s2o4 , uimm10s2o4) LC = PREG	LoopSetup	
--	10	lsetup (uimm10s2o4) LC = uimm10	LoopSetup	2.0
00	11	lsetup (uimm4s2o4 , uimm10s2o4) LC = PREG >> 1	LoopSetup	
01	01	lsetupz (uimm10s2o4) LC = PREG	LoopSetup	2.0
01	11	lsetupz (uimm10s2o4) LC = PREG >> 1	LoopSetup	2.0
10	01	lsetuplez (uimm10s2o4) LC = PREG	LoopSetup	2.0
10	11	lsetuplez (uimm10s2o4) LC = PREG >> 1	LoopSetup	2.0

LC

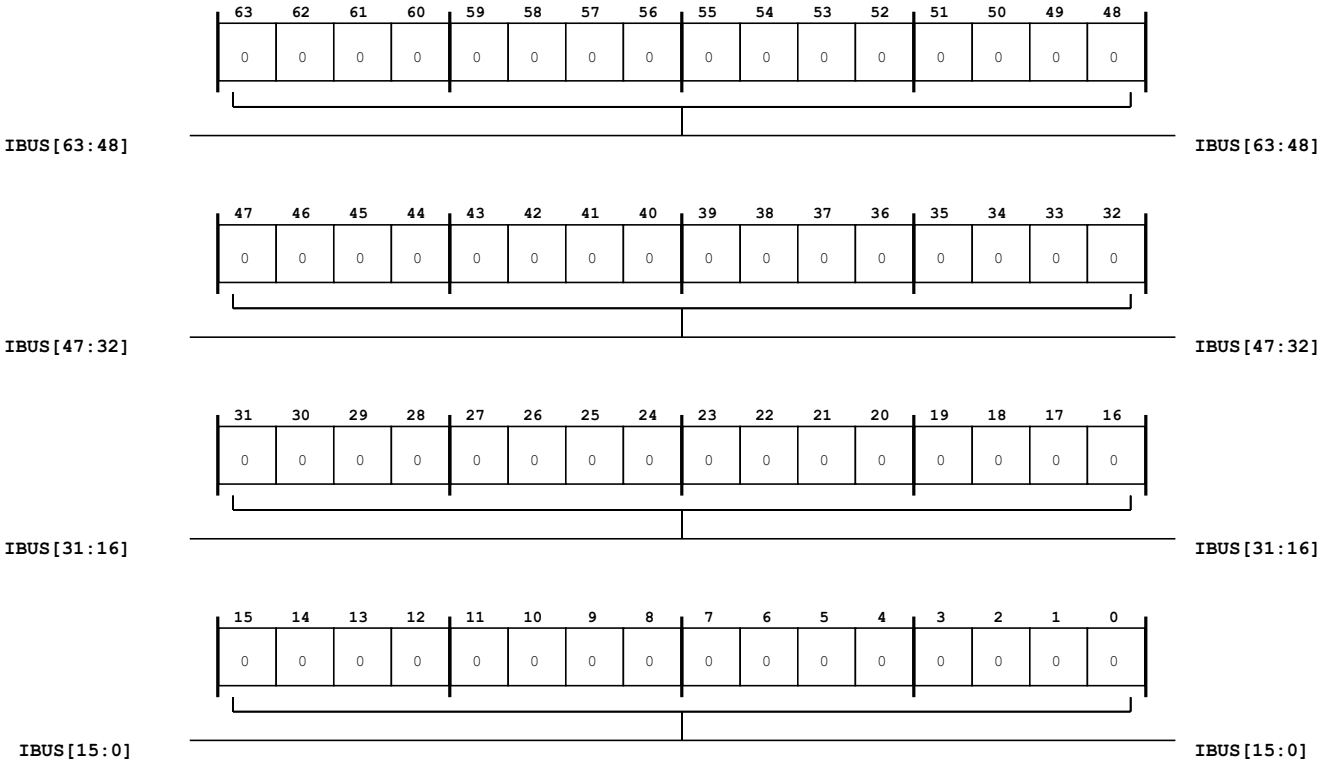
LC Encode Table

C	Syntax
0	lc0
1	lc1

64-bit Instruction Shell (Multi)

Multi Instruction Syntax

64-bit Instruction Shell (Multi)



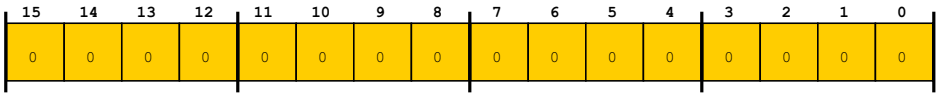
The following table provides the opcode field values (IBUS[63:59]), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

IBUS[63:59]	Syntax	Rev
0----	MAIN16A;	
10---	MAIN16B;	
110-0	MAIN32A;	
111--	MAIN32B;	
11011	MAIN64;	2.0
11001	SLOTM || SLOT0 || SLOT1;	

16-bit Slot Nop (NOP16)

NOP16 Instruction Syntax

16-bit Slot Nop (NOP16)



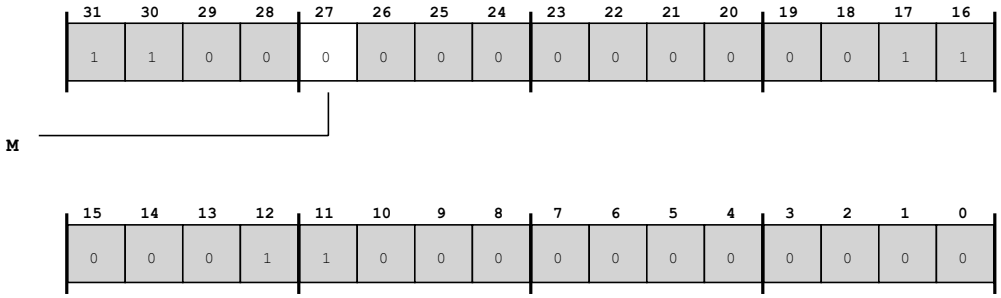
The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

	Syntax	Instruction
	nop	NOP

32-bit Slot Nop (NOP32)

NOP32 Instruction Syntax

32-bit Slot Nop (NOP32)



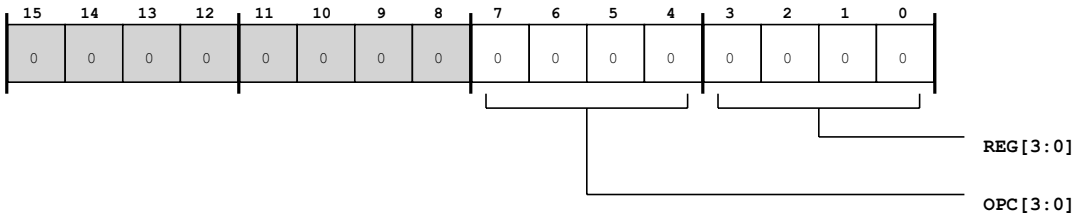
The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

	Syntax	Instruction
	mnop	NOP32

Basic Program Sequencer Control Functions (ProgCtrl)

ProgCtrl Instruction Syntax

Basic Program Sequencer Control Functions (ProgCtrl)



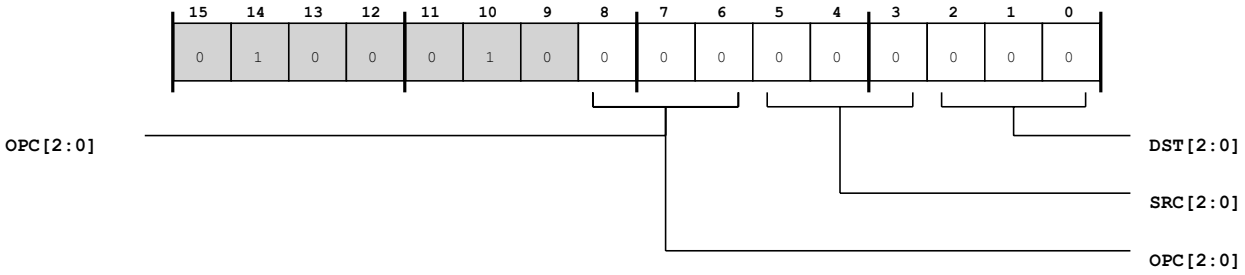
The following table provides the opcode field values (OPC, REG), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	REG	Syntax	Instruction	Rev
0001	0000	rts	Return	
0001	0001	rti	Return	
0001	0010	rtx	Return	
0001	0011	rtn	Return	
0001	0100	rte	Return	
0010	0000	idle	Sync	
0010	0011	csync	Sync	
0010	0100	ssync	Sync	
0010	0101	emuexcpt	Mode	
0011	0---	cli DREG	IMaskMv	
0100	0---	sti DREG	IMaskMv	
0101	0---	jump (PREG)	Jump	
0110	0---	call (PREG)	Call	
0111	0---	call (pc+ PREG)	Call	
1000	0---	jump (pc+ PREG)	Jump	
1001	----	raise uimm4	Raise	
1010	----	excpt uimm4	Raise	
1011	0---	testset (PREGP)	TestSet	
1100	0---	sti idle DREG	Sync	2.1.1

Pointer Arithmetic Operations (Ptr2op)

Ptr2op Instruction Syntax

Pointer Arithmetic Operations (Ptr2op)



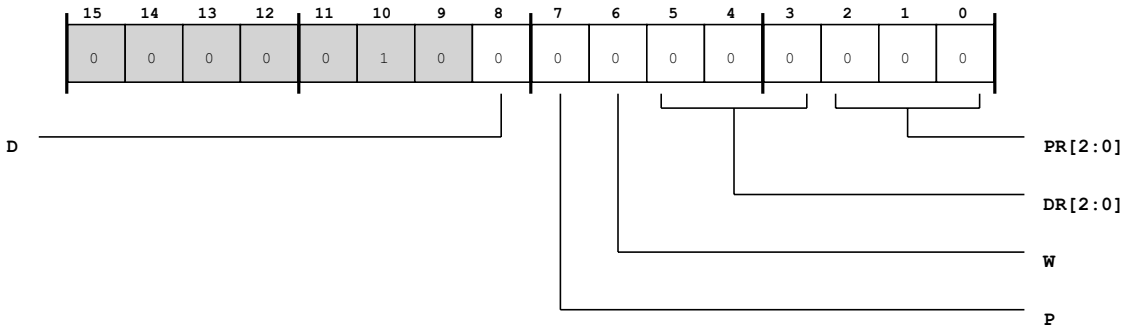
The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
000	PREG -= PREG	DagAdd32
001	PREG = PREG << 2	LShiftPtr
010	PREG = PREG << 1	LShiftPtr
011	PREG = PREG >> 2	LShiftPtr
100	PREG = PREG >> 1	LShiftPtr
101	PREG += PREG (brev)	DagAdd32
110	PREG =(PREG + PREG) << 1	DagAddSubShift
111	PREG =(PREG + PREG) << 2	DagAddSubShift

Push or Pop Multiple contiguous registers (Push-PopMult)

PushPopMult Instruction Syntax

Push or Pop Multiple contiguous registers (PushPopMult)



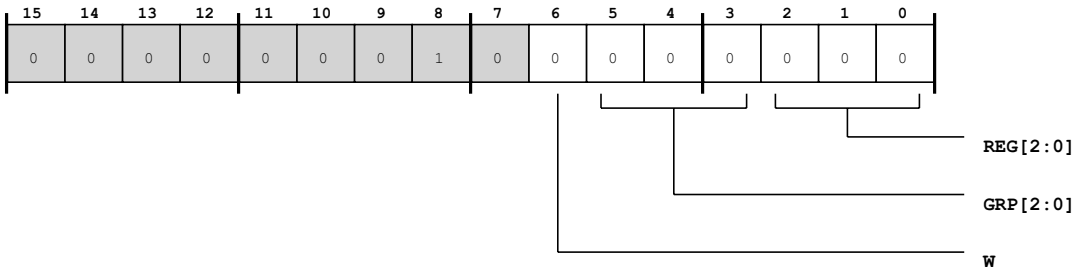
The following table provides the opcode field values (W, D, P), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	D	P	Syntax	Instruction
0	0	1	(PREG_RANGE) = [sp++]	PushPopMul16
0	1	0	(DREG_RANGE) = [sp++]	PushPopMul16
0	1	1	(DREG_RANGE , PREG_RANGE) = [sp++]	PushPopMul16
1	0	1	[--sp] = (PREG_RANGE)	PushPopMul16
1	1	0	[--sp] = (DREG_RANGE)	PushPopMul16
1	1	1	[--sp] = (DREG_RANGE , PREG_RANGE)	PushPopMul16

Push or Pop register, to and from the stack pointed to by sp (PushPopReg)

PushPopReg Instruction Syntax

Push or Pop register, to and from the stack pointed to by sp (PushPopReg)



The following table provides the opcode field values (W), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	Syntax	Instruction
0	POPREG = [sp++]	Pop
1	[--sp] = PUSHREG	Push

POPREG

POPREG Encode Table

GRP	REG	Syntax
010	0--	IREG
010	1--	MREG
011	0--	BREG
011	1--	LREG
100	000	a0.x
100	001	a0.w
100	010	a1.x
100	011	a1.w
100	110	astat

GRP	REG	Syntax
100	111	rets
110	---	SYSREG2
111	---	SYSREG3

PUSHREG

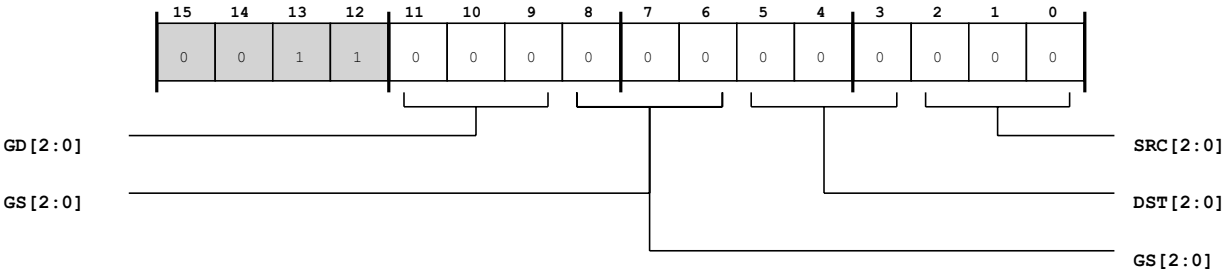
PUSHREG Encode Table

GRP	REG	Syntax
000	---	DREG
001	---	PREG
010	0--	IREG
010	1--	MREG
011	0--	BREG
011	1--	LREG
100	000	a0.x
100	001	a0.w
100	010	a1.x
100	011	a1.w
100	110	astat
100	111	rets
110	---	SYSREG2
111	---	SYSREG3

Register to register transfer operation (RegMv)

RegMv Instruction Syntax

Register to register transfer operation (RegMv)



The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

	Syntax	Instruction
	GDST = GSRC	MvRegToReg

GDST

GDST Encode Table

GD	DST	Syntax
000	---	DREG
001	---	PREG
010	0--	IREG
010	1--	MREG
011	0--	BREG
011	1--	LREG
100	000	a0.x
100	001	a0.w
100	010	a1.x
100	011	a1.w
100	110	astat

GD	DST	Syntax
100	111	rets
110	---	SYSREG2
111	---	SYSREG3

GSRC

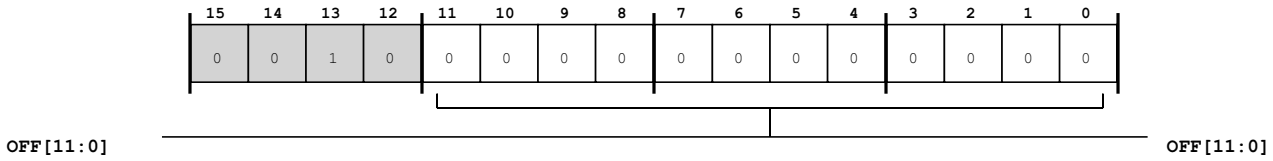
GSRC Encode Table

GS	SRC	Syntax
000	---	DREG
001	---	PREG
010	0--	IREG
010	1--	MREG
011	0--	BREG
011	1--	LREG
100	000	a0.x
100	001	a0.w
100	010	a1.x
100	011	a1.w
100	110	astat
100	111	rets
110	---	SYSREG2
111	---	SYSREG3

Unconditional Branch PC relative with 12bit offset (UJump)

UJump Instruction Syntax

Unconditional Branch PC relative with 12bit offset (UJump)



The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

	Syntax	Instruction
	jump.s imm12nxs2	JumpAbs

bimm32 Register Type

bimm32 Attributes

range	allow_label
-0x80000000:0x7fffffff	true

buimm32 Register Type

buimm32 Attributes

range	allow_label
0x0:0xffffffff	true

huimm16 Register Type

huimm16 Attributes

range
0x0:0xffff

imm10s2 Register Type

imm10s2 Attributes

range	allow_label
-0x400:0x3fe:2	true

imm12nxs2 Register Type

imm12nxs2 Attributes

range	allow_label
-0x1000:0xffe:2	true

imm12s2 Register Type

imm12s2 Attributes

range	allow_label
-0x1000:0xffe:2	true

imm12xs2 Register Type

imm12xs2 Attributes

range	allow_label
-0x1000:0xffe:2	true

imm16 Register Type

imm16 Attributes

range
-0x8000:0x7fff

imm16negpos Register Type

imm16negpos Attributes

range	negated
0x1:0x8000	true

imm16reloc Register Type

imm16reloc Attributes

range
-0x8000:0x7fff

imm16s2 Register Type

imm16s2 Attributes

range
-0x10000:0xfffe:2

imm16s2negpos Register Type

imm16s2negpos Attributes

range	negated
0x2:0x10000:2	true

imm16s4 Register Type

imm16s4 Attributes

range
-0x20000:0x1fffc:4

imm16s4negpos Register Type

imm16s4negpos Attributes

range	negated
0x4:0x20000:4	true

imm24nxs2 Register Type

imm24nxs2 Attributes

range
-0x1000000:0xfffffe:2

imm24s2 Register Type

imm24s2 Attributes

range
-0x1000000:0xfffffe:2

imm24xs2 Register Type

imm24xs2 Attributes

range
-0x1000000:0xfffffe:2

imm3 Register Type

imm3 Attributes

range
-0x4:0x3

imm32 Register Type

imm32 Attributes

range
-0x80000000:0x7fffffff

imm5nzs4negpos Register Type

imm5nzs4negpos Attributes

range	negated
0x4:0x80:4	true

imm6 Register Type

imm6 Attributes

range
-0x20:0x1f

imm7 Register Type

imm7 Attributes

range
-0x40:0x3f

luimm16 Register Type

luimm16 Attributes

range
0x0:0xffff

negimm5s4 Register Type

negimm5s4 Attributes

range
-0x80:-0x4:4

rimm16 Register Type

rimm16 Attributes

range
0x0:0xffff

uimm10 Register Type

uimm10 Attributes

range	iencode
0x1:0x3ff,0xffffffff	0xffffffff:0

uimm10s2o4 Register Type

uimm10s2o4 Attributes

range	allow_label
0x4:0x7fe:2	true

uimm16s4 Register Type

uimm16s4 Attributes

range
0x0:0x3fffc:4

uimm3 Register Type

uimm3 Attributes

range
0x0:0x7

uimm32 Register Type

uimm32 Attributes

range	allow_label
0x0:0xffffffff	true

uimm4 Register Type

uimm4 Attributes

range
0x0:0xf

uimm4nz Register Type

uimm4nz Attributes

range
0x1:0xf

uimm4nznegpos Register Type

uimm4nznegpos Attributes

range	negated
0x1:0xf	true

uimm4s2 Register Type

uimm4s2 Attributes

range
0x0:0x1e:2

uimm4s2o4 Register Type

uimm4s2o4 Attributes

range	allow_label
0x4:0x1e:2	true

uimm4s4 Register Type

uimm4s4 Attributes

range
0x0:0x3c:4

uimm5 Register Type

uimm5 Attributes

range
0x0:0x1f

uimm5nz Register Type

uimm5nz Attributes

range
0x1:0x1f

uimm5nznegpos Register Type

uimm5nznegpos Attributes

range	negated
0x1:0x1f	true

BREG Register Type

BREG Syntax

Code	Syntax
00	b0
01	b1
10	b2
11	b3

BREG_H Register Type

BREG_H Syntax

Code	Syntax
00	b0.h
01	b1.h
10	b2.h
11	b3.h

BREG_L Register Type

BREG_L Syntax

Code	Syntax
00	b0.l

Code	Syntax
01	b1.l
10	b2.l
11	b3.l

DREG Register Type

DREG Syntax

Code	Syntax
000	r0
001	r1
010	r2
011	r3
100	r4
101	r5
110	r6
111	r7

DREG_B Register Type

DREG_B Syntax

Code	Syntax
000	r0.b
001	r1.b
010	r2.b
011	r3.b
100	r4.b
101	r5.b
110	r6.b
111	r7.b

DREG_E Register Type

DREG_E Syntax

Code	Syntax
000	r0
010	r2
100	r4
110	r6

DREG_H Register Type

DREG_H Syntax

Code	Syntax
000	r0.h
001	r1.h
010	r2.h
011	r3.h
100	r4.h
101	r5.h
110	r6.h
111	r7.h

DREG_L Register Type

DREG_L Syntax

Code	Syntax
000	r0.l
001	r1.l
010	r2.l
011	r3.l
100	r4.l
101	r5.l

Code	Syntax
110	r6.l
111	r7.l

DREG_O Register Type

DREG_O Syntax

Code	Syntax
000	r1
010	r3
100	r5
110	r7

DREG_PAIR Register Type

DREG_PAIR Syntax

Code	Syntax
000	r1:0
010	r3:2
100	r5:4
110	r7:6

DREG_RANGE Register Type

DREG_RANGE Syntax

Code	Syntax
000	r7:0
001	r7:1
010	r7:2
011	r7:3

Code	Syntax
100	r7:4
101	r7:5
110	r7:6
111	r7:7

IREG Register Type

IREG Syntax

Code	Syntax
00	i0
01	i1
10	i2
11	i3

IREG_H Register Type

IREG_H Syntax

Code	Syntax
00	i0.h
01	i1.h
10	i2.h
11	i3.h

IREG_L Register Type

IREG_L Syntax

Code	Syntax
00	i0.l
01	i1.l

Code	Syntax
10	i2.l
11	i3.l

LREG Register Type

LREG Syntax

Code	Syntax
00	l0
01	l1
10	l2
11	l3

LREG_H Register Type

LREG_H Syntax

Code	Syntax
00	l0.h
01	l1.h
10	l2.h
11	l3.h

LREG_L Register Type

LREG_L Syntax

Code	Syntax
00	l0.l
01	l1.l
10	l2.l
11	l3.l

MREG Register Type

MREG Syntax

Code	Syntax
00	m0
01	m1
10	m2
11	m3

MREG_H Register Type

MREG_H Syntax

Code	Syntax
00	m0.h
01	m1.h
10	m2.h
11	m3.h

MREG_L Register Type

MREG_L Syntax

Code	Syntax
00	m0.l
01	m1.l
10	m2.l
11	m3.l

PREG Register Type

PREG Syntax

Code	Syntax
000	p0
001	p1
010	p2
011	p3
100	p4
101	p5
110	sp
111	fp

PREGP Register Type

PREGP Syntax

Code	Syntax
000	p0
001	p1
010	p2
011	p3
100	p4
101	p5

PREG_H Register Type

PREG_H Syntax

Code	Syntax
000	p0.h
001	p1.h
010	p2.h
011	p3.h

Code	Syntax
100	p4.h
101	p5.h
110	sp.h
111	fp.h

PREG_L Register Type

PREG_L Syntax

Code	Syntax
000	p0.l
001	p1.l
010	p2.l
011	p3.l
100	p4.l
101	p5.l
110	sp.l
111	fp.l

PREG_RANGE Register Type

PREG_RANGE Syntax

Code	Syntax
000	p5:0
001	p5:1
010	p5:2
011	p5:3
100	p5:4
101	p5:5

SYSREG2 Register Type

SYSREG2 Syntax

Code	Syntax
000	lc0
001	lt0
010	lb0
011	lc1
100	lt1
101	lb1
110	cycles
111	cycles2

SYSREG3 Register Type

SYSREG3 Syntax

Code	Syntax
000	usp
001	seqstat
010	syscfg
011	reti
100	retx
101	retn
110	rete
111	emudat

Issuing Parallel Instructions

This chapter discusses the instructions that can be issued in parallel. It identifies supported combinations for parallel issue, parallel issue syntax, 32-bit ALU/MAC instructions, 16-bit instructions, and examples.

The Blackfin processor is not superscalar; it does not execute multiple instructions at once. However, it does permit up to three instructions to be issued in parallel with some limitations. A multi-issue instruc-

tion is 64-bits in length and consists of one 32-bit instruction and two 16-bit instructions. All three instructions execute in the same amount of time as the slowest of the three.

Sections in this chapter

Supported Parallel Combinations

The diagram in Supported Parallel Combinations illustrates the combinations for parallel issue that the Blackfin processor supports.

32-bit ALU/MAC instruction	16-bit Instruction	16-bit Instruction

Parallel Issue Syntax

The syntax of a parallel issue instruction is as follows.

- *A 32-bit ALU/MAC instruction || A 16-bit instruction || A 16-bit instruction* ;

  The vertical bar (||) indicates the following instruction is to be issued in parallel with the previous instruction. Note the terminating semicolon appears only at the end of the parallel issue instruction.

  It is possible to issue a 32-bit ALU/MAC instruction in parallel with only one 16-bit instruction using the following syntax. The result is still a 64-bit instruction with a 16-bit NOP automatically inserted into the unused 16-bit slot.

- *A 32-bit ALU/MAC instruction || A 16-bit instruction* ;

  Alternately, it is also possible to issue two 16-bit instructions in parallel with one another without an active 32-bit ALU/MAC instruction by using the MNOP instruction, shown below. Again, the result is still a 64-bit instruction.

- MNOP || *A 16-bit instruction || A 16-bit instruction* ;

  See the MNOP (32-bit NOP) instruction description in NOP. The MNOP instruction does not have to be explicitly included by the programmer; the software tools prepend it automatically. The MNOP instruction will appear in disassembled parallel 16-bit instructions.

32-Bit ALU/MAC Instructions

The list of 32-bit instructions that can be in a parallel instruction are shown in the **32-Bit DSP Instructions** table.

Table 8-50: 32-Bit DSP Instructions

Instruction Name (Description)	Operation Type and Parallel Version Notes
Arithmetic Operations	
Abs32 (Absolute Value)	Dsp32Alu
AddSub32 (Add or Subtract)	Dsp32Alu Note: Only permits parallelism for versions supporting saturation.
AddSubRnd12 (Add/Subtract – Prescale Up)	Dsp32Alu
AddSubRnd20 (Add/Subtract – Prescale Down)	Dsp32Alu
Shift_ExpAdj32 (Exponent Detection)	Dsp32Shf
Max32 (Maximum)	Dsp32Alu
Min32 (Minimum)	Dsp32Alu
AddSubAcc (Modify Increment/Decrement)	Dsp32Alu
Neg32 (Negate, Two's-Complement)	Dsp32Alu Note: Only permits parallelism for the accumulator versions.
Pass32Rnd16 (Round to Half-Word)	Dsp32Alu
ALU_SatAcc0 (Saturate A0)	Dsp32Alu
ALU_SatAcc1 (Saturate A1)	Dsp32Alu
ALU_SatAccDual (Dual Saturate)	Dsp32Alu
Shift_SignBits32	Dsp32Shf
Load Store	
LdImmToAx (Clear A0)	Dsp32Alu
LdImmToAx (Clear A1)	Dsp32Alu
LdImmToAxDual (Dual Clear)	Dsp32Alu
Bit Operations	
Shift_Deposit (Bit Field Deposit)	Dsp32Shf
Shift_Extract (Bit Field Extract)	Dsp32Shf
BitMux (Bit Multiplex)	Dsp32Shf
Shift_Ones (Ones Count)	Dsp32Shf
Logical Operations	
BXORShift_NF ((Bitwise XOR with Feedback)	Dsp32Shf
BXOR_NF (Bitwise XOR without Feedback)	Dsp32Shf

Table 8-50: 32-Bit DSP Instructions (Continued)

Instruction Name (Description)	Operation Type and Parallel Version Notes
Move	
MvDregToAx (Move Register to A0)	Dsp32Alu
MvDregToAx (Move Register to A1)	Dsp32Alu
MvA0ToDregE (Move A0.X to Register Half)	Dsp32Alu
MvA1ToDreg0 (Move A1.X to Register Half)	Dsp32Alu
MvDregHLToAxHL (Move Register Half to A0)	Dsp32Alu
MvDregHLToAxHL (Move Register Half to A1)	Dsp32Alu
Shift / Rotate Operations[1]	
AShiftAcc (Arithmetic Shift A0/A1)	Dsp32Shf Note: See footnote restrictions.
AShift32 (Arithmetic Shift Register)	Dsp32Shf Note: Only permits parallelism for saturating versions; see footnote restrictions.
LShift16 (Logical Shift Half Register by Half Register or Immediate)	Dsp32Shf and Dsp32Shf Note See footnote restrictions.
Shift_Rot32 (Rotate Register)	Dsp32Shf Note: See footnote restrictions.
Shift_RotAcc (Rotate A0/A1)	Dsp32Shf Note: See footnote restrictions.
External Event Management	
NOP32 (No Operation)	NOP32 Note: Only permits parallelism for 32-bit MNOP.
Vector Operations	
Shift_VitMax (Modulo Maximum with History)	Dsp32Shf
AddOnSign (Add on Sign)	Dsp32Alu
Mac16 (Multiply to Accumulator)	Dsp32Mult
Mac16 (Multiply-Accumulate to Accumulator)	Dsp32Mult
Mult16 (Multiply to Half Register)	Dsp32Mult
Mac16 (Multiply-Accumulate to Half Register)	Dsp32Mac
Mult32 (Multiply to Register)	Dsp32Mult
Mac16 (Multiply-Accumulate to Register)	Dsp32Mult
Abs2x16 (Absolute Value, Vector)	Dsp32Alu
AddSubVec16 (Add or Subtract, Vector)	Dsp32Alu
AShift16Vec (Arithmetic Shift Register, Vector)	Dsp32Shf

Table 8-50: 32-Bit DSP Instructions (Continued)

Instruction Name (Description)	Operation Type and Parallel Version Notes
LShift16Vec (Logical Shift Register by Half Register or Immediate, Vector)	Dsp32Shf
Max16Vec (Maximum, Vector)	Dsp32Alu
Min16Vec (Minimum, Vector)	Dsp32Alu
Mult16 (Multiply 16-Bit Operands)	Dsp32Mult
Neg16Vec (Negate, Two's-Complement, Vector)	Dsp32Alu
Pack16Vec (Pack, Vector)	Dsp32Shf
Search (Search, Vector)	Dsp32Alu
Video Pixel Operations	
Shift_Align (Byte Align 8, 16, and 24)	Dsp32Shf
DisAlignExcept (Disable Alignment Exception for Load)	Dsp32Alu
SAD8Vec (Sum of Absolute Differences, Vector)	Dsp32Alu
AddAccHalf (Dual Half Register Add to A0/A1)	Dsp32Alu
AddSub4x8 (Quad 8-Bit Add/Subtract)	Dsp32Alu
Avg8Vec (Quad 8-Bit Average - Byte)	Dsp32Alu
Avg4x8Vec (Quad 8-Bit Average - Half Word)	Dsp32Alu
AddClip (Dual 16-Bit Add/Clip)	Dsp32Alu
BytePack (Quad 8-Bit Pack)	Dsp32Alu
ByteUnPack (Quad 8-Bit Unpack)	Dsp32Alu

1. Multi-issue may not combine SHIFT/ROTATE with STORE using Preg + Offset operation.

16-Bit Instructions

The two 16-bit instructions in a multi-issue instruction must each be from the instructions shown in the **Compatible 16-Bit Instructions** table.

The following additional restrictions also apply to the 16-bit instructions of the multi-issue instruction.

- Only one of the 16-bit instructions can be a store instruction.
- Only one of the 16-bit instructions may load a pointer register. This load must be encoded in DAG slot 0.

Table 8-51: Compatible 16-Bit Instructions

Instruction Name (Description)	Operation Type and Parallel Version Notes	
Arithmetic Operations		

Table 8-51: Compatible 16-Bit Instructions (Continued)

Instruction Name (Description)	Operation Type and Parallel Version Notes	
DagAddImm (DAG Add/Subtract Immediate)	CompI2opP Note: I-Register versions only.	
DagAdd32 (DAG Modify Increment/Decrement)	Ptr2op Note: I-Register versions only.	
Load / Store		
LdImmToReg (Load Pointer Register)	Ldp , LdpII , and LdStIdxI	
LdM32bitToDreg (Load Data Register)	LdStPmod , DspLdSt , LdSt , LdStIIFP , LdpIIFP , LdStII , LdStIdxI , and LdStAbs	
LdM16bitToDreg (Load Half Word Sign/Zero Extended)	LdStPmod , LdSt , LdStII , LdStIdxI , and LdStAbs	
LdM16bitToDregH (Load High Half Register)	LdStPmod , DspLdSt , and LdStAbs	
LdM16bitToDregL (Load Low Half Register)	LdStPmod , DspLdSt , and LdStAbs	
LdM08bitToDreg (Load Byte Sign/Zero Extended)	LdSt , LdStIdxI , and LdStAbs	
StPregToM32bit (Store Pointer Register)	LdSt	
StDregToM32bit (Store Data Register)	LdStPmod , DspLdSt , LdSt , LdStIIFP , LdStII , LdStIdxI , and LdStAbs	
StDregHToM16bit (Store High Half Data Register)	LdStPmod , DspLdSt , and LdStAbs	
StDregLToM16bit (Store Low Half Data Register)	LdStPmod , DspLdSt , LdSt , LdStII , LdStIdxI , and LdStAbs	
StDregToM08bit (Store Byte)	LdSt , LdStIdxI , and LdStAbs	
External Event Management		
NOP (No Operation)	NOP16 Note: 16-bit NOP only.	

Parallel Operation Examples

Two Parallel Memory Access Instructions

```
/* Subtract-Absolute-Accumulate issued in parallel with the memory access instructions
that fetch the data for the next SAA instruction. This sequence is executed in a loop
to flip-flop back and forth between the data in R1 and R3, then the data in R0 and R2.
*/
saa (r1:0, r3:2) || r0=[i0++] || r2=[i1++] ;
```

```
saa (r1:0, r3:2)(r) || r1=[i0++] || r3=[i1++] ;
mnop || r1 = [i0++] || r3 = [i1++] ;
```

One *Ireg* and One Memory Access Instruction in Parallel

```
/* Add on Sign while incrementing an Ireg and loading a data register based on the
previous value of the Ireg. */
r7.h=r7.l=sign(r2.h)*r3.h + sign(r2.l)*r3.l || i0+=m3 || r0=[i0] ;
/* Add/subtract two vector values while incrementing an Ireg and loading a data
register. */
R2 = R2 +|+ R4, R4 = R2 -|- R4 (ASR) || I0 += M0 (BREV) || R1 = [I0] ;
/* Multiply and accumulate to Accumulator while loading a data register and storing a
data register using an Ireg pointer. */
A1=R2.L*R1.L, A0=R2.H*R1.H || R2.H=W[I2++] || [I3++]=R3 ;
/* Multiply and accumulate while loading two data registers. One load uses an Ireg
pointer. */
A1+=R0.L*R2.H,A0+=R0.L*R2.L || R2.L=W[I2++] || R0=[I1--] ;
R3.H=(A1+=R0.L*R1.H), R3.L=(A0+=R0.L*R1.L) || R0=[P0++] || R1=[I0] ;
/* Pack two vector values while storing a data register using an Ireg pointer and
loading another data register. */
R1=PACK(R1.H,R0.H) || [I0++]=R0 || R2.L=W[I2++] ;
```

One *Ireg* Instruction in Parallel

```
/* Multiply-Accumulate to a Data register while incrementing an Ireg. */
r6=(a0+=r3.h*r2.h)(fu) || i2-=m0 ;
/* which the assembler expands into:
   r6=(a0+=r3.h*r2.h)(fu) || i2-=m0 || nop ; */
```

9 Debug

The Blackfin processor's debug functionality is used for software debugging. It also complements some services often found in an operating system (OS) kernel. The functionality is implemented in the processor hardware and is grouped into multiple levels.

A summary of available debug features is shown in the **Blackfin Debug Features** table.

Table 9-1: Blackfin Debug Features

Debug Feature	Description
Watchpoints	Specify address ranges and conditions that halt the processor when satisfied.
Trace History	Stores the last 16 discontinuous values of the Program Counter in an on-chip trace buffer.
Cycle Count	Provides functionality for all code profiling functions.
Performance Monitoring	Allows internal resources to be monitored and measured non-intrusively.

Watchpoint Unit

By monitoring the addresses on both the instruction bus and the data bus, the Watchpoint Unit provides several mechanisms for examining program behavior. After counting the number of times a particular address is matched, the unit schedules an event based on this count.

In addition, information that the Watchpoint Unit provides helps in the optimization of code. The unit also makes it easier to maintain executables through code patching.

The Watchpoint Unit contains these memory-mapped registers (MMRs), which are accessible in Supervisor and Emulator modes:

- The Watchpoint Status register (`WPSTAT`)
- Six Instruction Watchpoint Address registers (`WPIA[5:0]`)
- Six Instruction Watchpoint Address Count registers (`WPIACNT[5:0]`)
- The Instruction Watchpoint Address Control register (`WPIACTL`)
- Two Data Watchpoint Address registers (`WPDA[1:0]`)
- Two Data Watchpoint Address Count registers (`WPDACNT[1:0]`)

- The Data Watchpoint Address Control register (WPDACTL)

Two operations implement instruction watchpoints:

- The values in the six Instruction Watchpoint Address registers, WPIA[5:0], are compared to the address on the instruction bus.
- Corresponding count values in the Instruction Watchpoint Address Count registers, WPIACNT[5:0], are decremented on each match.

The six Instruction Watchpoint Address registers may be further grouped into three ranges of instruction-address-range watchpoints. The ranges are identified by the addresses in WPIA0 to WPIA1, WPIA2 to WPIA3, and WPIA4 to WPIA5.

- WPIA0 <= WPIA1
- WPIA2 <= WPIA3
- WPIA4 <= WPIA5

Two operations implement data watchpoints:

- The values in the two Data Watchpoint Address registers, WPDA[1:0], are compared to the address on the data buses.
- Corresponding count values in the Data Watchpoint Address Count registers, WPDACNT[1:0], are decremented on each match.

The two Data Watchpoint Address registers may be further grouped together into one data-address-range watchpoint, WPDA[1:0].

The instruction and data count value registers must be loaded with the number of times the watchpoint must match minus one. After the count value reaches zero, the subsequent watchpoint match results in an exception or emulation event.

An event can also be triggered on a combination of the instruction and data watchpoints. If the WPAND bit in the WPIACTL register is set, then an event is triggered only when both an instruction address watchpoint matches *and* a data address watchpoint matches. If the WPAND bit is 0, then an event is triggered when any of the enabled watchpoints or watchpoint ranges match.

To enable the Watchpoint Unit, the WPPWR bit in the WPIACTL register must be set. If WPPWR = 1, then the individual watchpoints and watchpoint ranges may be enabled using the specific enable bits in the WPIACTL and WPDACTL MMRs. If WPPWR = 0, then all watchpoint activity is disabled.

Instruction Watchpoints

Each instruction watchpoint is controlled by three bits in the WPIACTL register, as shown in the **WPIACTL Control Bits** table.

Table 9-2: WPIACTL Control Bits

Bit Name	Description
EMUSWx	Determines whether an instruction-address match causes either an emulation event or an exception event.
WPICNTENx	Enables the 16-bit counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPIAENx	Enables the address watchpoint activity.

When two watchpoints are associated to form a range, two additional bits are used, as shown in the **WPIACTL Watchpoint Range Control Bits** table.

Table 9-3: WPIACTL Watchpoint Range Control Bits

Bit Name	Description
WPIRENxy	Indicates the two watchpoints that are to be associated to form a range.
WPIRINVxy	Determines whether an event is caused by an address within the range identified or outside of the range identified.

Code patching allows software to replace sections of existing code with new code. The watchpoint registers are used to trigger an exception at the start addresses of the earlier code. The exception routine then vectors to the location in memory that contains the new code.

On the processor, code patching can be achieved by writing the start address of the earlier code to one of the `WPIAx` registers and setting the corresponding `EMUSWx` bit to trigger an exception. In the exception service routine, the `WPSTAT` register is read to determine which watchpoint triggered the exception. Next, the code writes the start address of the new code in the `RETX` register, and then returns from the exception to the new code. Because the exception mechanism is used for code patching, event service routines of the same or higher priority (exception, NMI, and reset routines) cannot be patched.

A write to the `WPSTAT` MMR clears all the sticky status bits. The data value written is ignored.

WPIAx Registers

When the Watchpoint Unit is enabled, the values in the Instruction Watchpoint Address registers (`WPIAx`) are compared to the address on the instruction bus. Corresponding count values in the Instruction Watchpoint Address Count registers (`WPIACNTx`) are decremented on each match. For more information, see Watchpoint Instruction Address Register.

Register Name	Memory-Mapped Address
WPIA0	0xFFE0 7040
WPIA1	0xFFE0 7044

Register Name	Memory-Mapped Address
WPIA2	0xFFE0 7048
WPIA3	0xFFE0 704C
WPIA4	0xFFE0 7050
WPIA5	0xFFE0 7054

WPIACNTx Registers

When the Watchpoint Unit is enabled, the count values in the Instruction Watchpoint Address Count registers (`WPIACNT[5:0]`) are decremented each time the address or the address bus matches a value in the `WPIAx` registers. Load the `WPIACNTx` register with a value that is one less than the number of times the watchpoint must match before triggering an event. The `WPIACNTx` register will decrement to 0x0000 when the programmed count expires. For more information, see the Watchpoint Instruction Address Count Register.

Register Name	Memory-Mapped Address
WPIACNT0	0xFFE0 7080
WPIACNT1	0xFFE0 7084
WPIACNT2	0xFFE0 7088
WPIACNT3	0xFFE0 708C
WPIACNT4	0xFFE0 7090
WPIACNT5	0xFFE0 7094

WPIACTL Register

Three bits in the Instruction Watchpoint Address Control register (`WPIACTL`) control each instruction watchpoint. For more information about the bits in this register, see Watchpoint Unit and Watchpoint Instruction Address Control Register 01.

Data Address Watchpoints

Each data watchpoint is controlled by four bits in the `WPDACTL` register, as shown in the **Data Address Watchpoints** table.

Table 9-4: Data Address Watchpoints

Bit Name	Description
WPDACCx	Determines whether the match should be on a read or write access.

Table 9-4: Data Address Watchpoints (Continued)

Bit Name	Description
WPDSRCx	Determines which DAG the unit should monitor.
WPDCNTENx	Enables the counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPDAENx	Enables the data watchpoint activity.

When the two watchpoints are associated to form a range, two additional bits are used. See the **WPDACTL Watchpoint Control Bits** table.

Table 9-5: WPDACTL Watchpoint Control Bits

Bit Name	Description
WPDREN01	Indicates the two watchpoints associated to form a range.
WPDRINV01	Determines whether an event is caused by an address within the range identified or outside the range.

WPDAx Registers

When the Watchpoint Unit is enabled, the values in the Data Watchpoint Address registers (`WPDAx`) are compared to the address on the data buses. Corresponding count values in the Data Watchpoint Address Count registers (`WPDACNTx`) are decremented on each match. For more information, see the Watchpoint Data Address Register.

WPDACNTx Registers

When the Watchpoint Unit is enabled, the count values in the Data Watchpoint Address Count Value registers (`WPDACNTx`) are decremented each time the address or the address bus matches a value in the `WPDAx` registers. Load this `WPDACNTx` register with a value that is one less than the number of times the watchpoint must match before triggering an event. The `WPDACNTx` register will decrement to 0x0000 when the programmed count expires. For more information, see the Watchpoint Data Address Count Value Register.

WPDACTL Register

For more information about the bits in the Data Watchpoint Address Control register (`WPDACTL`), see Data Address Watchpoints and Watchpoint Data Address Control Register.

WPSTAT Register

The Watchpoint Status register (WPSTAT) monitors the status of the watchpoints. It may be read and written in Supervisor or Emulator modes only. When a watchpoint or watchpoint range matches, this register reflects the source of the watchpoint. The status bits in the WPSTAT register are sticky, and all of them are cleared when any write, regardless of the value, is performed to the register. For more information, see the Watchpoint Status Register.

Performance Monitor Unit

The Blackfin architecture provides a built-in performance monitor unit (PMU) to non-intrusively monitor the processor's internal resources. The PMU includes a set of processor events that can be counted during program execution.

A subset of these processor events can be counted in terms of the *number of stalls* that occur while the event is active or true, in units of core clock cycles. This stall measurement provides an indication of the performance penalty associated with the event. The rest of the processor events can be counted in terms of the *number of occurrences* of the event. This event measurement helps with program debugging and provides an aid to understanding the performance bottlenecks in an application.

Developers can use the PMU to count pipeline and memory stalls. The stall information can be used iteratively to find quickly areas on which to focus during the optimization process. The debugging efficiency comes from using the PMU when running applications directly on hardware as oppose to predicting these events in a simulation environment.

For example, the PMU can help to detect whether the performance bottleneck is due to L1 data memory access latency. One can then find out, using another PMU event, whether the memory stall is due to a core and DMA access to the L1 memory. The processor core and DMA controller can access different sub banks of memory in the same cycle, but when these resources attempt to access the same sub bank in the same cycle, one of the accesses must stall. (Refer to *Overview of On-Chip Level 1 (L1) Memory* in the Memory chapter for more details on L1 memory arbitration stalls). After finding these issues with the PMU, one could iteratively move buffers to non-conflicting banks of the L1 memory to minimize the core and DMA access conflicts.

Functional Description

The PMU provides two sets of registers (PFCTRx and PFCTL), which permit non-intrusive monitoring of the processor's internal resources during program execution.

The performance monitor counter (PFCNTR1-0) registers are 32-bit registers that hold the number of occurrences of a selected event from within a processor core. Each of the counters must be enabled prior to use.

The performance control (PFCTL) register provides:

- Enable/disable of the performance monitoring unit,
- Selection of the *event mode*,
- Configuration of the *event type* to be monitored, and
- Selection of interrupt handling type for a counter overflow condition.

Together, these registers provide feedback indicating the measure of load balancing between the various resources on the chip. This feedback permits comparison and analysis of expected versus actual resource usage.

PFCNTRx Registers

The **Performance Monitor Counter Registers** figure shows the performance monitor counter registers, PFCNTR[1:0]. The PFCNTR0 register contains the count value of performance counter 0. The PFCNTR1 register contains the count value of performance counter 1. For more information, see Counter 0 Register and Counter 1 Register.

The counter retains its value even after the module is disabled, so the programmer has to clear the counter before using it again. The counter can also be programmed with a non-zero 32 bit value.

PFCTL Register

To enable the PMU, set the PFPWR bit in the performance monitor control register (PFCTL). After the unit is enabled, individual count-enable bits (PFCENx) take effect. Use the PFCENx bits to enable or disable the performance monitors in User mode, Supervisor mode, or both. Use the PEMUSWx bits to select the type of event triggered. For more information, see the Control Register.

PFCNTx - Event Mode

Setting the PFCNT1-0 bits enable the events that are listed in the **PFMONx Event Type (Occurrences)** table (see PFMON - Event Type) to be counted as an "occurrence" of an event and clearing these bits enables the events listed in the **PFMONx Event Type (Stalls)** table (see PFMON - Event Type) to be counted as "number of stalls" while the event is active or true.

PFMON - Event Type

Use the PEMON7-0 bits to select the type of event triggered.

The **PFMONx Event Type (Occurrences)** table identifies events that cause the performance monitor counter registers (PFMON0 or PFMON1) to increment based on the number of "occurrences". For the events listed in the **PFMONx Event Type (Occurrences)** table, set the PFCONTx bit.

Table 9-6: PFMONx Event Type (Occurrences)

PFMONx fields	Events Incrementing Count Based on Number-of-Occurrences
0x04	PC invariant branches (Jump address that can be determined solely from the instruction.
0x06	Number of branch mispredictions
0x09	Branches "taken". Includes calls, returns, branches, but not interrupts. It excludes branches taken but mispredicted.
0x0B	EXCPT instruction
0x0C	CSYNC or an SSYNC instruction
0x0D	Instructions committed to the core
0x0E	Interrupts taken (includes exceptions and emulator breakpoints)
0x0F	Misaligned address violation exceptions
0x80	Code memory fetches postponed due to DMA collisions (minimum count of two per event)
0x83	Code memory 64-bit words delivered to processor instruction assembly Unit
0x9A	Data memory cache fills completed to Bank A
0x9B	Data memory cache fills completed to Bank B
0x9C	Data memory cache victims delivered from Bank A
0x9D	Data memory cache victims delivered from Bank B
0x9E	Data memory cache high priority fills requested
0x9F	Data memory cache low priority fills requested

The **PFMONx Event Type (Stalls)** table identifies events that cause the performance monitor counter registers (`PFMON0` or `PFMON1`) to increment based on the "number of stalls" until the event is true. For the events listed in the **PFMONx Event Type (Stalls)** table, clear the `PFCONTx` bit.

Table 9-7: PFMONx Event Type (Stalls)

PFMONx fields	Events Incrementing Count Based on Number-of-Stalls (until the event is active)
0x0	Loop 0 iterations (loop 0 counter decrements)
0x01	Loop 1 iterations (loop 1 counter decrements)
0x0A	Stalls due to CSYNC and SSYNC instructions
0x10	Stalls due to read after write hazards on DAG registers
0x13	Stalls due to RAW data hazards in computes
0x81	Code memory TAG stalls (cache misses, or FlushI operations, count of 3 per FlushI). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.

Table 9-7: PFMONx Event Type (Stalls) (Continued)

PFMONx fields	Events Incrementing Count Based on Number-of-Stalls (until the event is active)
0x82	Code memory fill stalls (cacheable or non-cacheable). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.
0x90	Processor stalls to memory
0x91	Data memory stalls to processor not hidden by processor stall
0x92	Data memory store buffer full stalls
0x93	Data memory write buffer full stalls due to high-to-low priority code transition
0x95	Data memory fill buffer stalls
0x96	Data memory array or TAG collision stalls (DAG to DAG, or DMA to DAG)
0x97	Data memory array collision stalls (DAG to DAG or DMA to DAG)
0x98	Data memory stalls
0x99	Data memory stalls sent to processor

PEMUSWx - Handling Counter Overflow Condition

The PMU can optionally be configured to generate either a hardware error or an emulation event when it rolls over based on the configuration of the `PEMUSWx` bit. When a hardware error is generated, the`HWERR-CAUSE1-0` bits of the `SEQSTAT` register are set to the value 0x12. The PMU can also be used to detect an instance of any of the events in the **PFMONx Event Type (Occurrences)** table (see PFMON - Event Type). To do this, the counter can be pre-loaded with the maximum value. The first time a selected event happens, a hardware error occurs to indicate that the event triggered.

Programming Example

The following code example demonstrates a possible use case of the PMU.

```
I0.L = LO(0xFF801004);
  /* L1 data memory address */
I0.H = HI(0xFF801004);
I1.L = LO(0xFF801244);
  /* L1 data memory address in same 4K sub-bank */
I1.H = HI(0xFF801244);
P0.L = LO(PFCTL);
P0.H = HI(PFCTL);
R0 = 0;
[P0] = R0;
  /* reset performance control register */
P0.L = LO(PFCNTR0);
P0.H = HI(PFCNTR0);
R0 = 0;
[P0] = R0;
```

```
  /* reset performance counter 0 */
P0.L = LO(PFCTL);
P0.H = HI(PFCTL);
R0.L = 0x0019;
  /* enable the monitor and counter 0 */
R0.H = 0x0000;
R1 = PFCEN_VALUE;
  /* load the event number (0x96) */
R0 = R0 | R1;
[P0] = R0;
  /* program performance control register */
R1 = R4.L * R5.H (IS) ||
  R3 = [I0++] ||
  R4 = [I1++];
  /* parallel instruction accessing
     2 data memory locations */
```

This code example would result in the increment of the counter for a value of 1. This is the indication of one stall. This stall is due to a collision in the data bank a, sub-bank 1).

A one cycle stall is incurred during a collision of simultaneous accesses only if the accesses are to the same 32-bit word polarity (address bits 2 match), the same 4 KB sub-bank (address bits 13 and 12 match), the same 16 KB half-bank (address bits 16 match), and the same bank (address bits 21 and 20 match).

The hardware-error interrupt can be made use of in cases where it is required to have an indication for specific counter increments. For this, the PEMUSWx bit has to be cleared and the counter has to be pre-loaded with a value of 0xFFFFFFFF.

```
P0.L = LO(PFCNTR0);
P0.H = HI(PFCNTR0);
R0.L = 0xFFFF;
R0.H = 0xFFFF;
[P0] = R0;
```

Because the PEMUSW0 bit is cleared, the counter overflow would result in a hardware error interrupt.

```
P0.L = LO(IMASK);
  /* LOAD IMASK ADDRESS */
P0.H = HI(IMASK);
R0 = [P0];
R1 = IVHW;
  /* UNMASK HARDWARE ERROR INTERRUPT */
R0 = R0 | R1;
[P0] = R0;
P0.L = LO(EVT5);
  /* STORE ISR HANDLER ADDRESS */
P0.H = HI(EVT5);
R0.L = LO(IVHW_ISR);
R0.H = HI(IVHW_ISR);
[P0] = R0;
IVHW_ISR:
P0 = SEQSTAT;
  /* LOAD SEQUENCER STATUS REGISTER VALUE */
```

```
R0 = [P0];
R1 = 0x12;
  /* CHECK FOR HWERRCAUSE 0x12 */
R1 <<= 14;
CC = R1 == R0;
IF !CC JUMP EXIT;
PFMON_OVERFLOW:
  /* AN OVERLFOW HAS BEEN DETECTED */
  /* *************** */
  /* Performance Monitor overflow detected */

  /* user code */
  /* --------- */
  RTI;
```

Cycle Counter

The cycle counter counts CCLK cycles while the program is executing. All cycles, including execution, wait state, interrupts, and events, are counted while the processor is in User or Supervisor mode, but the cycle counter stops counting in Emulator mode.

The cycle counter is 64 bits and increments every cycle. The count value is stored in two 32-bit registers, CYCLES and CYCLES2. The least significant 32 bits (LSBs) are stored in CYCLES. The most significant 32 bits (MSBs) are stored in CYCLES2.

The CYCLES and CYCLES2 registers are read/write in all modes (User, Supervisor, and Emulator modes) for all Blackfin processors. For more information, see Cycle Count (32 LSBs) Register and Cycle Count (32 MSBs) Register.

To enable the cycle counters, set the CCEN bit in the SYSCFG register. The following example shows how to use the cycle counter:

```
R2 = 0;
CYCLES = R2;
CYCLES2 = R2;
R2 = SYSCFG;
BITSET(R2,1);
SYSCFG = R2;
/* Insert code to be benchmarked here. */
R2 = SYSCFG;
BITCLR(R2,1);
SYSCFG = R2;
```

CYCLES and CYCLES2 Registers

The Execution Cycle Count registers (CYCLES and CYCLES2) form a 64-bit counter that increments every CCLK cycle. The CYCLES register contains the least significant 32 bits of the cycle counter's 64-bit count value. The most significant 32 bits are contained by CYCLES2. For more information, see Cycle Count (32

LSBs) Register and Cycle Count (32 MSBs) Register.

Note when single-stepping through instructions in a debug environment, the CYCLES register increases in non-unity increments due to the interaction of the debugger over JTAG.

SYSCFG Register

The System Configuration register (SYSCFG) controls the configuration of the processor. This register is accessible only from the Supervisor mode. For more information, see the System Configuration Register.

Product Identification Register

The 32-bit DSP Device ID register (DSPID) is a core MMR that contains core identification and revision fields for the core.

DSPID Register

The DSP Device ID register (DSPID) is a read-only register and is part of the processor core. This register differs depending on whether the processor has a single core or is a a dual-core processor. For more information, see the DSP Identification Register.

ADSP-BF70x DBG Register Descriptions

Debug Unit (DBG) contains the following registers.

Table 9-8: ADSP-BF70x DBG Register List

Name	Description
DSPID	DSP Identification Register

DSP Identification Register

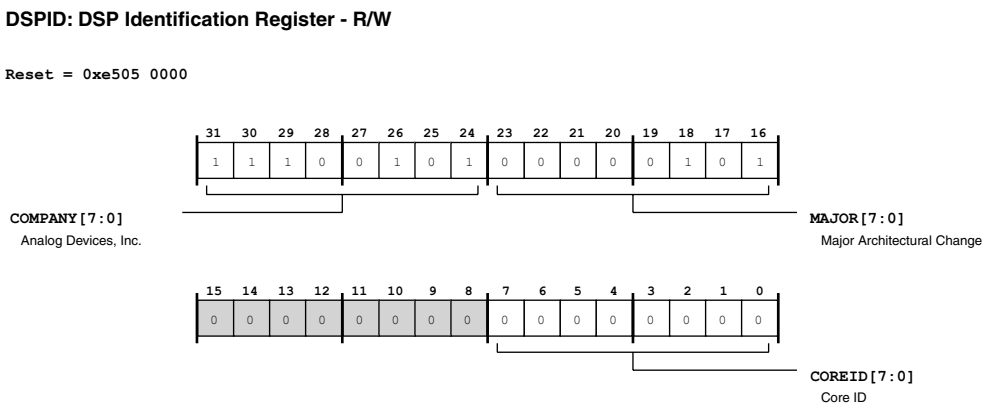


Figure 9-1: DSPID Register Diagram

Table 9-9: DSPID Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:24 (R/NW)	COMPANY	Analog Devices, Inc.	
		229	Analog Devices, Inc.
23:16 (R/NW)	MAJOR	Major Architectural Change.	
		4	ADSP-BF533
		5	Blackfin 2.0
7:0 (R/NW)	COREID	Core ID.	

ADSP-BF70x WP Register Descriptions

Watchpoint Unit (WP) contains the following registers.

Table 9-10: ADSP-BF70x WP Register List

Name	Description
WPIACTL	Watchpoint Instruction Address Control Register 01
WPIAn	Watchpoint Instruction Address Register
WPIACNTn	Watchpoint Instruction Address Count Register
WPDACTL	Watchpoint Data Address Control Register
WPDAn	Watchpoint Data Address Register
WPDACNTn	Watchpoint Data Address Count Value Register

Table 9-10: ADSP-BF70x WP Register List (Continued)

Name	Description
WPSTAT	Watchpoint Status Register

Watchpoint Instruction Address Control Register 01

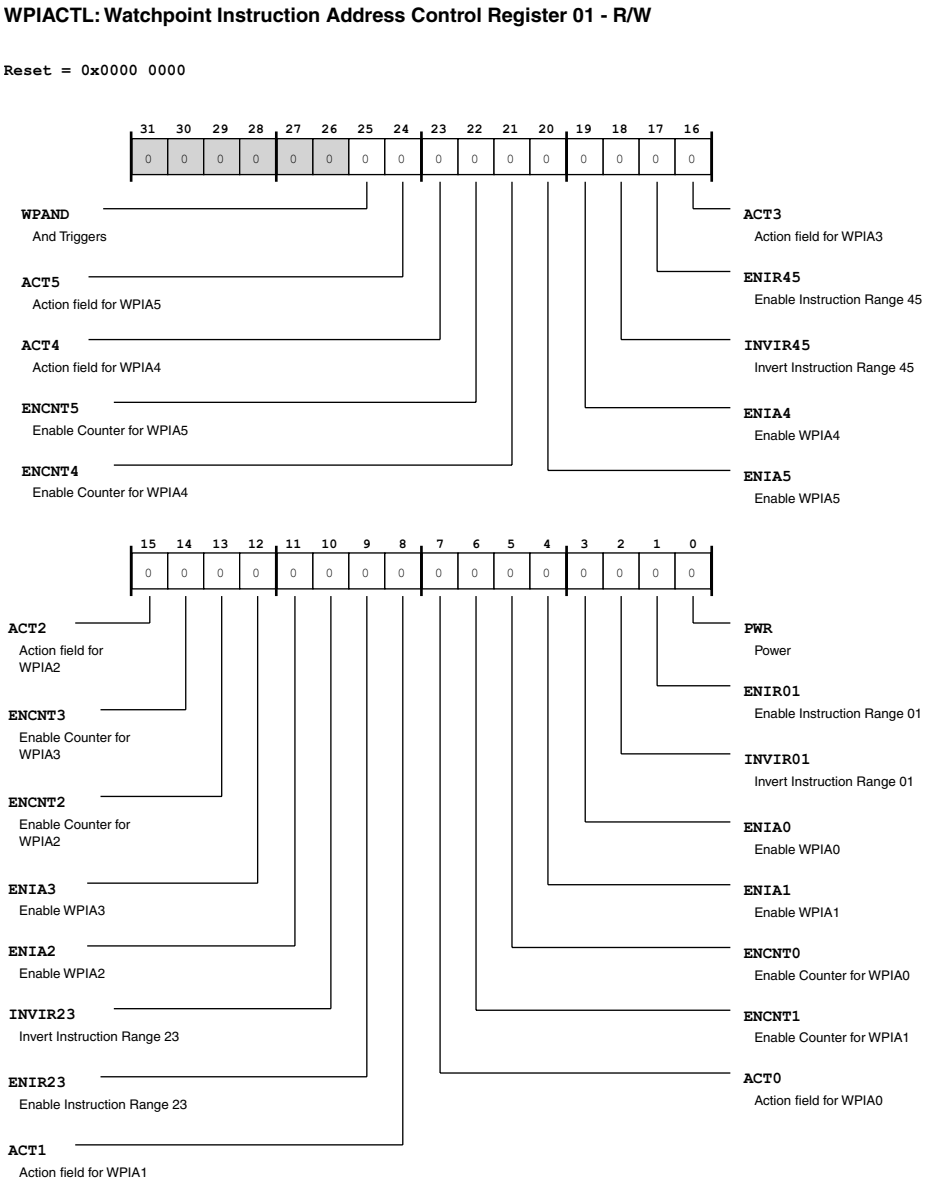


Figure 9-2: WPIACTL Register Diagram

Table 9-11: WPIACTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
25 (R/W)	WPAND	And Triggers.	
		0	Any enabled watchpoint triggers the action
		1	An enabled Inst AND data watchpoint trigger the action
24 (R/W)	ACT5	Action field for WPIA5.	
		0	WPIA5 match causes an exception event
		1	WPIA5 match causes an emulation event
23 (R/W)	ACT4	Action field for WPIA4.	
		0	WPIA4 match causes an exception event
		1	WPIA4 match causes an emulation event
22 (R/W)	ENCNT5	Enable Counter for WPIA5.	
		0	Disabled
		1	Enabled
21 (R/W)	ENCNT4	Enable Counter for WPIA4.	
		0	Disabled
		1	Enabled
20 (R/W)	ENIA5	Enable WPIA5.	
		0	Disabled
		1	Enabled
19 (R/W)	ENIA4	Enable WPIA4.	
		0	Disabled
		1	Enabled
18 (R/W)	INVIR45	Invert Instruction Range 45.	
17 (R/W)	ENIR45	Enable Instruction Range 45.	
		0	Disable Range
		1	Enable Range
16 (R/W)	ACT3	Action field for WPIA3.	
		0	WPIA3 match causes an exception event
		1	WPIA3 match causes an emulation event
15 (R/W)	ACT2	Action field for WPIA2.	
		0	WPIA2 match causes an exception event
		1	WPIA2 match causes an emulation event

Table 9-11: WPIACTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
14 (R/W)	ENCNT3	Enable Counter for WPIA3.	
		0	Disabled
		1	Enabled
13 (R/W)	ENCNT2	Enable Counter for WPIA2.	
		0	Disabled
		1	Enabled
12 (R/W)	ENIA3	Enable WPIA3.	
		0	Dialed
		1	Enabled
11 (R/W)	ENIA2	Enable WPIA2.	
		0	Disabled
		1	Enabled
10 (R/W)	INVIR23	Invert Instruction Range 23.	
9 (R/W)	ENIR23	Enable Instruction Range 23.	
		0	Disable Range
		1	Enable Range
8 (R/W)	ACT1	Action field for WPIA1.	
		0	WPIA1 match causes an exception event
		1	WPIA1 match causes an emulation event
7 (R/W)	ACT0	Action field for WPIA0.	
		0	WPIA0 match causes an exception event
		1	WPIA0 match causes an emulation event
6 (R/W)	ENCNT1	Enable Counter for WPIA1.	
		0	Disabled
		1	Enabled
5 (R/W)	ENCNT0	Enable Counter for WPIA0.	
		0	Disabled
		1	Enabled
4 (R/W)	ENIA1	Enable WPIA1.	
		0	Disabled
		1	Enabled
3 (R/W)	ENIA0	Enable WPIA0.	
		0	Disabled
		1	Enabled

Table 9-11: WPIACTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
2 (R/W)	INVIR01	Invert Instruction Range 01.	
1 (R/W)	ENIR01	Enable Instruction Range 01.	
		0	Disable Range
		1	Enable Range
0 (R/W)	PWR	Power. Address matches if it satisfies this equation WPIA0 < ADDRESS <= WPIA1	
		0	Disabled
		1	Enabled

Watchpoint Instruction Address Register

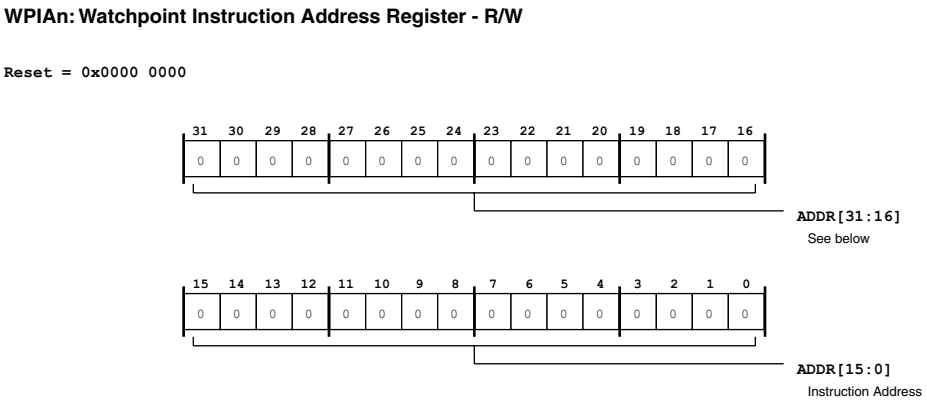


Figure 9-3: WPIAn Register Diagram

Table 9-12: WPIAn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Instruction Address.

Watchpoint Instruction Address Count Register

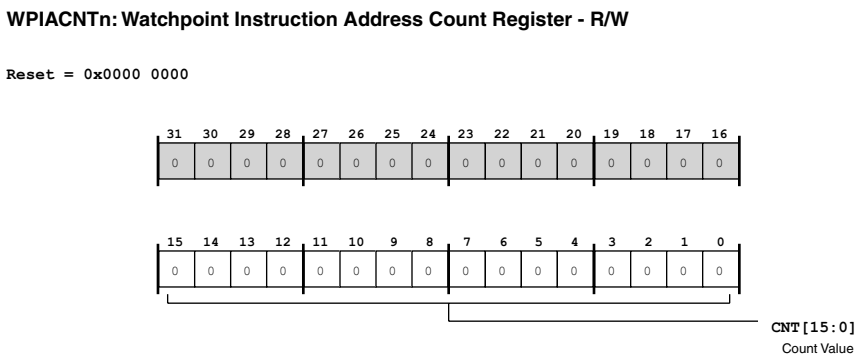


Figure 9-4: WPIACNTn Register Diagram

Table 9-13: WPIACNTn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	CNT	Count Value.

Watchpoint Data Address Control Register

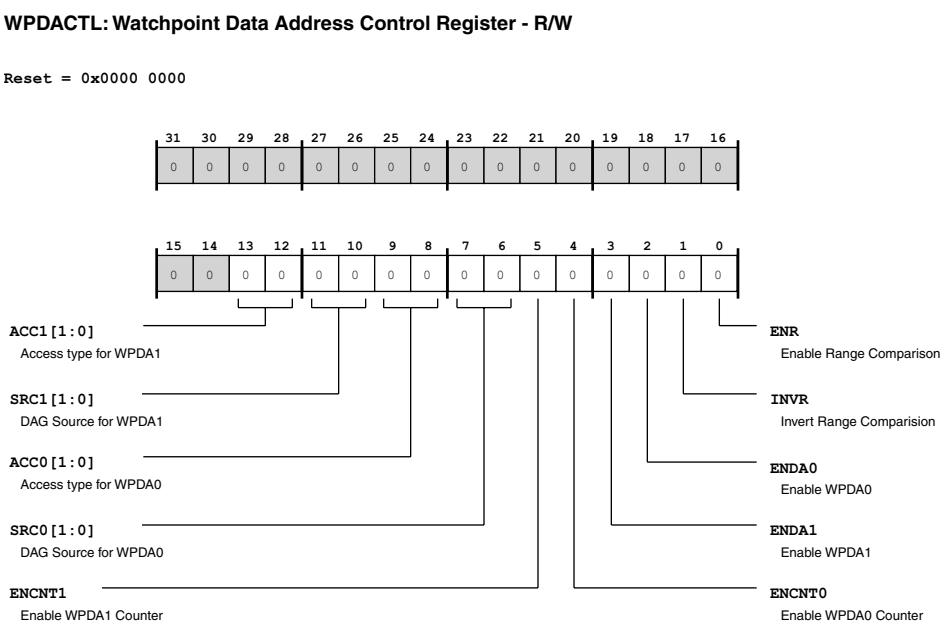


Figure 9-5: WPDACTL Register Diagram

Table 9-14: WPDACTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
13:12 (R/W)	ACC1	Access type for WPDA1.	
		0	Reserved
		1	Watch Writes only
		2	Watch Reads only
		3	Watch Reads and Writes
11:10 (R/W)	SRC1	DAG Source for WPDA1.	
		0	Reserved
		1	Watch DAG0
		2	Watch DAG1
		3	Watch Both DAGs
9:8 (R/W)	ACC0	Access type for WPDA0.	
		0	Reserved
		1	Watch Writes only
		2	Watch Reads only
		3	Watch Reads and Writes
7:6 (R/W)	SRC0	DAG Source for WPDA0.	
		0	Reserved
		1	Watch DAG0
		2	Watch DAG1
		3	Watch Both DAGs
5 (R/W)	ENCNT1	Enable WPDA1 Counter.	
4 (R/W)	ENCNT0	Enable WPDA0 Counter.	
3 (R/W)	ENDA1	Enable WPDA1.	
2 (R/W)	ENDA0	Enable WPDA0.	
1 (R/W)	INVR	Invert Range Comparison.	
0 (R/W)	ENR	Enable Range Comparison. Address matches if it satisfies this equation WPDA0 < ADDRESS <= WPDA1	

Watchpoint Data Address Register

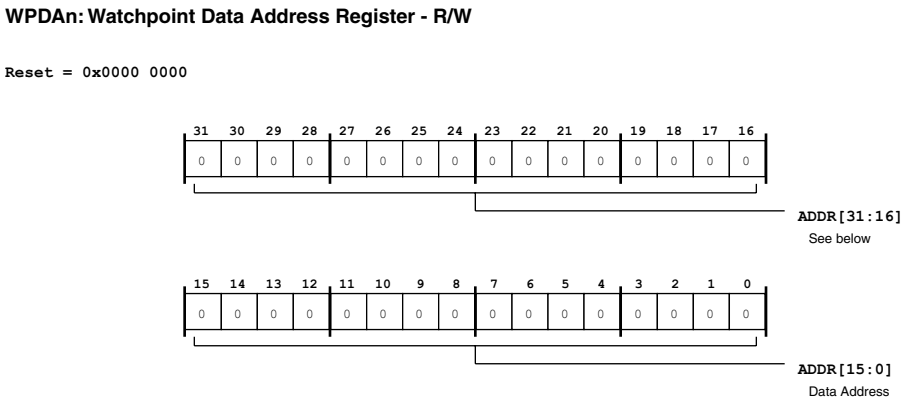


Figure 9-6: WPDAn Register Diagram

Table 9-15: WPDAn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Data Address.

Watchpoint Data Address Count Value Register

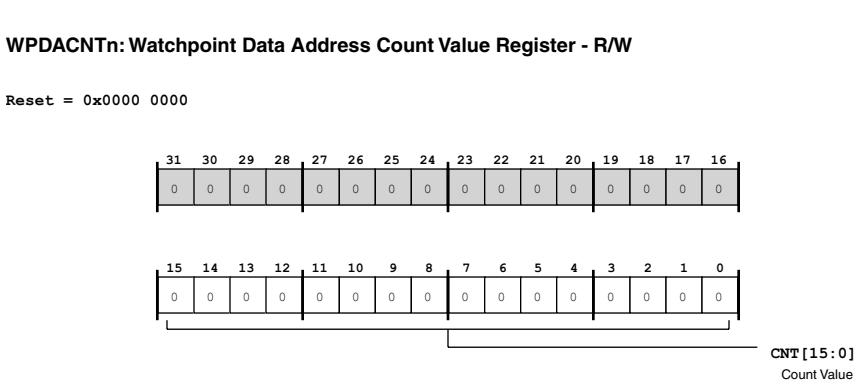


Figure 9-7: WPDACNTn Register Diagram

Table 9-16: WPDACNTn Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	CNT	Count Value.

Watchpoint Status Register

The Watchpoint Status register (WPSTAT) monitors the status of the watchpoints. It may be read and written in Supervisor or Emulator modes only. When a watchpoint or watchpoint range matches, this register reflects the source of the watchpoint. The status bits in the WPSTAT register are sticky, and all of them are cleared when any write, regardless of the value, is performed to the register.

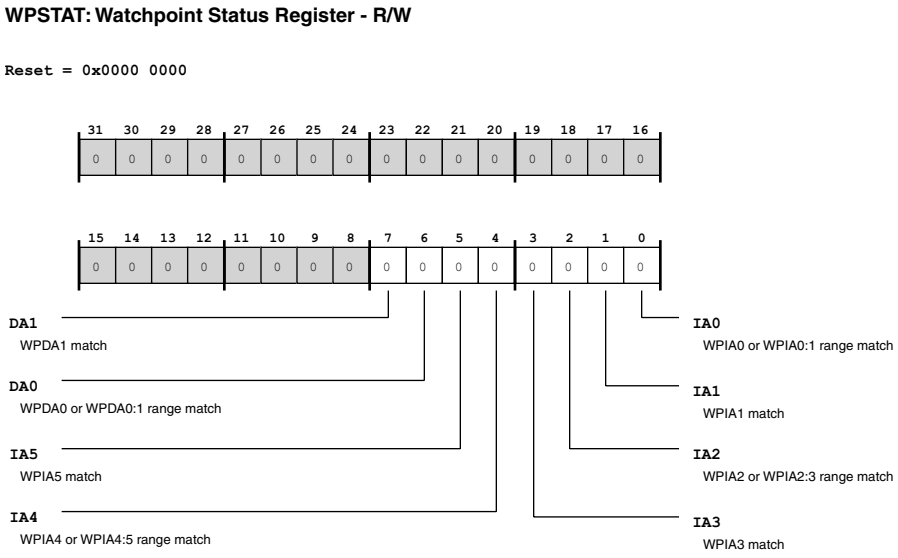


Figure 9-8: WPSTAT Register Diagram

Table 9-17: WPSTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
7 (R/W)	DA1	WPDA1 match.
6 (R/W)	DA0	WPDA0 or WPDA0:1 range match.
5 (R/W)	IA5	WPIA5 match.
4 (R/W)	IA4	WPIA4 or WPIA4:5 range match.

Table 9-17: WPSTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
3 (R/W)	IA3	WPIA3 match.
2 (R/W)	IA2	WPIA2 or WPIA2:3 range match.
1 (R/W)	IA1	WPIA1 match.
0 (R/W)	IA0	WPIA0 or WPIA0:1 range match.

ADSP-BF70x PF Register Descriptions

Performance Monitor (PF) contains the following registers.

Table 9-18: ADSP-BF70x PF Register List

Name	Description
PFCTL	Control Register
PFCNTR0	Counter 0 Register
PFCNTR1	Counter 1 Register

Control Register

The PFCTL register enables the performance monitor unit PF, selects whether event count expirations generate emulator or exception events, select the processor modes in which monitoring is enabled, and select the event type occurrences or stalls that the monitor counts.

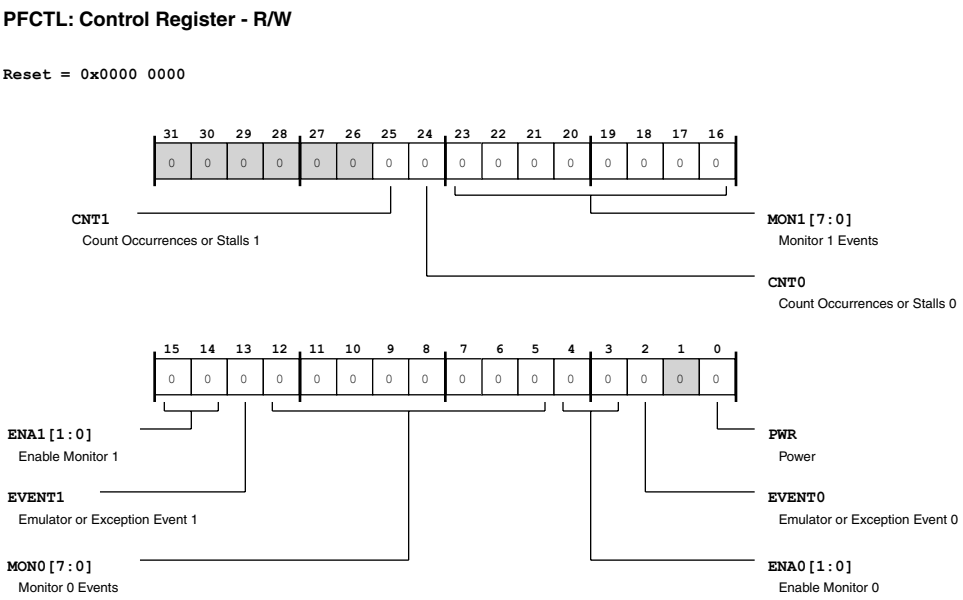


Figure 9-9: PFCTL Register Diagram

Table 9-19: PFCTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
25 (R/W)	CNT1	Count Occurrences or Stalls 1. The PFCTL.CNT1 bit selects whether monitor 1 counts the number of event type occurrences or event type related stall cycles for the event type selected with the PFCTL.MON1 bits.	
		0	Count stall cycles due to event type 1
		1	Count occurrences of event type 1
24 (R/W)	CNT0	Count Occurrences or Stalls 0. The PFCTL.CNT0 bit selects whether monitor 1 counts the number of event type occurrences or event type related stall cycles for the event type selected with the PFCTL.MON0 bits.	
		0	Count stall cycles due to event type 0
		1	Count occurrences of event type 0
23:16 (R/W)	MON1	Monitor 1 Events. The PFCTL.MON1 bits select the event type that is monitored, causing the PFCNTR1 count to decrement. The PFCTL.CNT1 bit selects whether it is occurrences of this type of event or stall cycles related to this type of event that affect the count. For information about event type values for the PFCTL.MON1 bit field, see the functional description of the PF unit.	

Table 9-19: PFCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
15:14 (R/W)	ENA1	Enable Monitor 1. The PFCTL.ENA1 bits select in which processor modes (user, supervisor, or both) the performance monitor 1 is enabled.	
		0	Disable monitor 1
		1	Enable monitor 1 in user mode only
		2	Enable monitor 1 in supervisor mode only
		3	Enable monitor 1 in user and supervisor mode
13 (R/W)	EVENT1	Emulator or Exception Event 1. The PFCTL.EVENT1 bit selects whether expiration of the PFCNTR1 count down causes an emulation event or an exception.	
		0	Exception on expired count 1
		1	Emulation event on expired count 1
12:5 (R/W)	MON0	Monitor 0 Events. The PFCTL.MON0 bits select the event type that is monitored, causing the PFCNTR0 count to decrement. The PFCTL.CNT0 bit selects whether it is occurrences of this type of event or stall cycles related to this type of event that affect the count. For information about event type values for the PFCTL.MON0 bit field, see the functional description of the PF unit.	
4:3 (R/W)	ENA0	Enable Monitor 0. The PFCTL.ENA0 bits select in which processor modes (user, supervisor, or both) the performance monitor 0 is enabled.	
		0	Disable monitor 0
		1	Enable monitor 0 in user mode only
		2	Enable monitor 0 in supervisor mode only
		3	Enable monitor 0 in user and supervisor mode
2 (R/W)	EVENT0	Emulator or Exception Event 0. The PFCTL.EVENT0 bit selects whether expiration of the PFCNTR0 count down causes an emulation event or an exception.	
		0	Exception on expired count 0
		1	Emulation event on expired count 0
0 (R/W)	PWR	Power. The PFCTL.PWR bit enables the PF.	
		0	Disable
		1	Enable

Counter 0 Register

The PFCNTR0 register holds the count value for performance monitor counter 0. Depending on the configuration of the PFCTL register, this count decrements based on monitored occurrences of events or stall

cycles related to events. When this count decrements to zero (expires), the PF issues an exception or an emulation event.

The PFCNTR0 counter retains its value even after the PF is disabled, so the counter must be cleared before it may be used again.

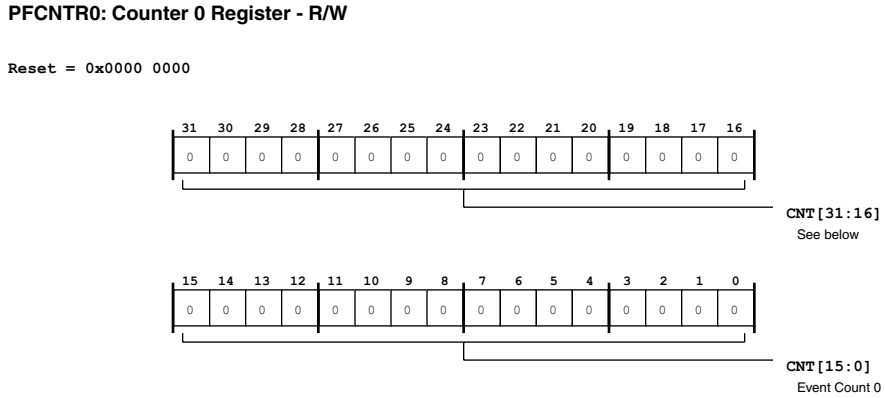


Figure 9-10: PFCNTR0 Register Diagram

Table 9-20: PFCNTR0 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	Event Count 0. The PFCNTR0.CNT bits hold the count value for performance monitor counter 0.

Counter 1 Register

The PFCNTR1 register holds the count value for performance monitor counter 1. Depending on the configuration of the PFCTL register, this count decrements based on monitored occurrences of events or stall cycles related to events. When this count decrements to zero (expires), the PF issues an exception or an emulation event.

The PFCNTR1 counter retains its value even after the PF is disabled, so the counter must be cleared before it may be used again.

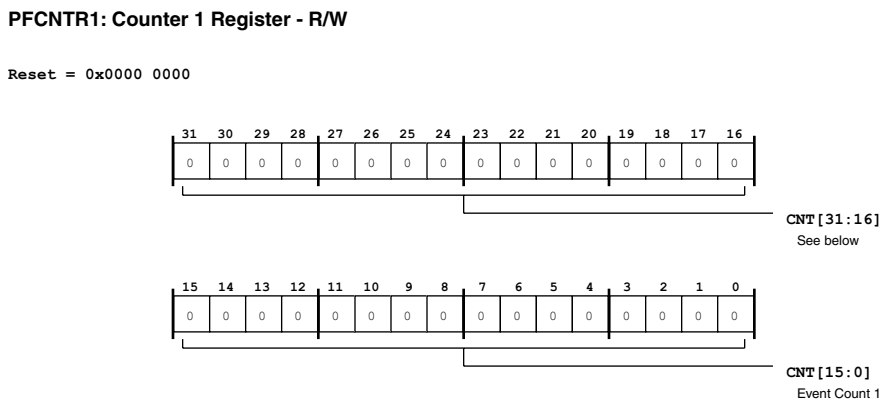


Figure 9-11: PFCNTR1 Register Diagram

Table 9-21: PFCNTR1 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	Event Count 1. The PFCNTR1.CNT bits hold the count value for performance monitor counter 1.

ADSP-BF70x Debug-Related (REGFILE) Register Descriptions

Register File (REGFILE) contains the following registers.

Table 9-22: ADSP-BF70x Debug-Related (REGFILE) Register List

Name	Description
SYSCFG	System Configuration Register
CYCLES	Cycle Counter Register
CYCLES2	Cycle Counter 2 Register

System Configuration Register

The SYSCFG register controls the configuration of the processor. This register is accessible only from the supervisor mode.

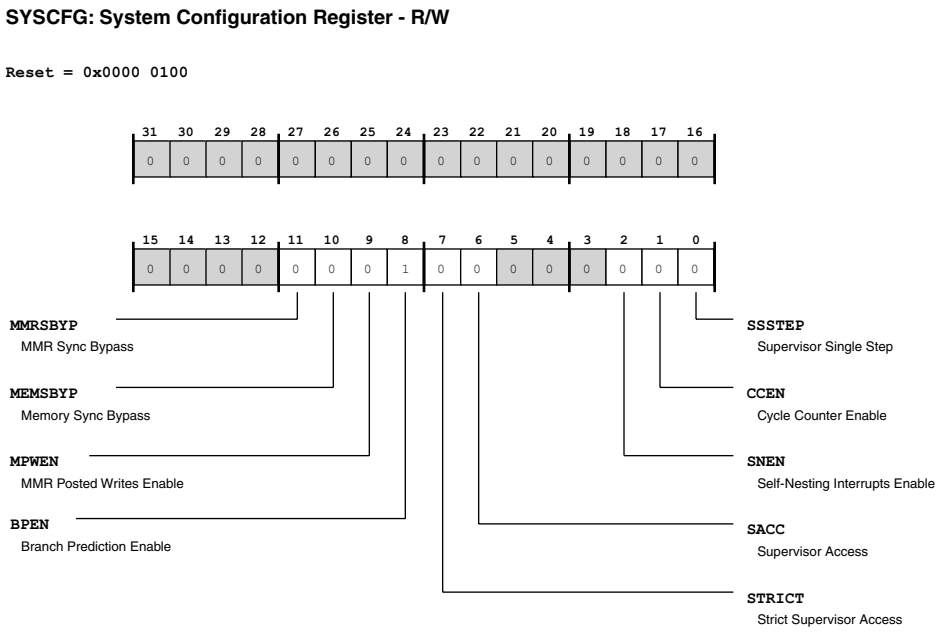


Figure 9-12: SYSCFG Register Diagram

Table 9-23: SYSCFG Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
11 (R/W)	MMRSBYP	MMR Sync Bypass. The SYSCFG.MMRSBYP bit enables bypass mode for clock domain synchronization in memory mapped register interface. Enabling this feature reduces read latency.	
		0	No bypass
		1	Bypass
10 (R/W)	MEMSBYP	Memory Sync Bypass. The SYSCFG.MEMSBYP bit enables bypass mode for clock domain synchronization in system memory bus interface. Enabling this feature reduces read latency.	
		0	No bypass
		1	Bypass
9 (R/W)	MPWEN	MMR Posted Writes Enable. The SYSCFG.MPWEN bit enables support for posting consecutive system MMR writes to the system fabric. When disabled, the processor waits for each response before allowing a new write to go into the system. Immediately after changing this mode (either enabling or disabling), an SSYNC instruction must be executed to allow the system to finish any outstanding writes before changing the rules on how writes go out to the system.	
		0	Disable
		1	Enable

Table 9-23: SYSCFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
8 (R/W)	BPEN	Branch Prediction Enable. The SYSCFG.BPEN bit selects whether the program sequencer uses dynamic or static branch prediction operation. For more information, see the Branch Prediction section of the Program Sequencer chapter.	
		0	Use static prediction
		1	Use dynamic prediction
7 (R/W)	STRICT	Strict Supervisor Access. The SYSCFG.STRICT bit restricts additional resources (beyond those restricted on the ADSP-BF5xx/BF6xx Blackfin processors) to require supervisor mode access. When enabled, accessing any of these additional resources in user mode (for example, executing an IDLE instruction) causes an illegal supervisor access exception. For more information, see the Protected Resources and Instructions section in the Operating Modes and States chapter.	
		0	Disable (normal/previous Blackfin access operation)
		1	Enable (strict supervisor access operation)
6 (R/W)	SACC	Supervisor Access. The SYSCFG.SACC bit selects whether supervisor mode access is permitted when the processor is not servicing any events or whether supervisor mode only is permitted when the processor is servicing an event.	
		0	Disable (access only when servicing an event)
		1	Enable (access whether or not servicing any events)
2 (R/W)	SNEN	Self-Nesting Interrupts Enable. The SYSCFG.SNEN bit enables self-nesting interrupt operation. Unlike interrupts during normal operation, interrupts during self-nesting may be interrupted by events at the same priority level. Self-nesting is supported for any interrupt level generated with the RAISE instruction, as well as for core level interrupts.	
		0	Disable (normal interrupt operation)
		1	Enable (self-nesting interrupt operation)
1 (R/W)	CCEN	Cycle Counter Enable. The SYSCFG.CCEN bit enables cycle counter operation. When the cycle counter is enabled, it counts core clock (CCLK) cycles, incrementing with each cycle. All cycles are counted (including wait states) in both user mode and supervisor modes. The cycle counter stops counting while the processor is in emulator mode. The cycle counter is 64 bits wide, and the count is stored in the CYCLES and CYCLES2 registers. The least significant 32 bits are stored in CYCLES.	
		0	Disable cycle counter
		1	Enable cycle counter

Table 9-23: SYSCFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
0 (R/W)	SSSTEP	Supervisor Single Step. The SYSCFG.SSSTEP bit enables single step operation, in which a supervisor exception occurs after the processor executes each instruction. This bit only applies to executing instructions in user mode or to processing interrupts in supervisor mode. The SYSCFG.SSSTEP bit is ignored if the core is processing an exception or higher-priority event. If precise exception timing is required, a CSYNC instruction must be used after setting this bit.	
		0	Disable (normal operation)
		1	Enable (single step operation)

Cycle Count (32 LSBs) Register

The CYCLES register holds the 32 least significant bits of the cycle count. The counter is enabled using the SYSCFG.CCEN bit. For more information, see the SYSCFG.CCEN bit description.

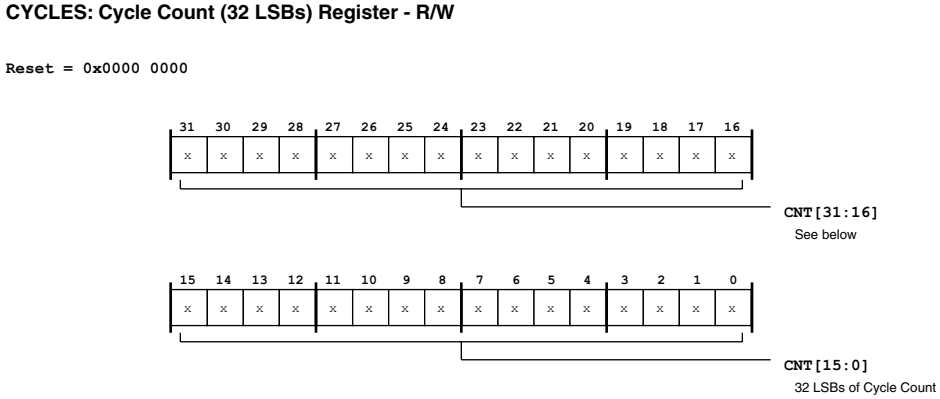


Figure 9-13: CYCLES Register Diagram

Table 9-24: CYCLES Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	32 LSBs of Cycle Count. The CYCLES.CNT bits hold the least significant 32 bits of the cycle count value.

Cycle Count (32 MSBs) Register

The CYCLES2 register holds the 32 most significant bits of the cycle count. The counter is enabled using the SYSCFG.CCEN bit. For more information, see the SYSCFG.CCEN bit description.

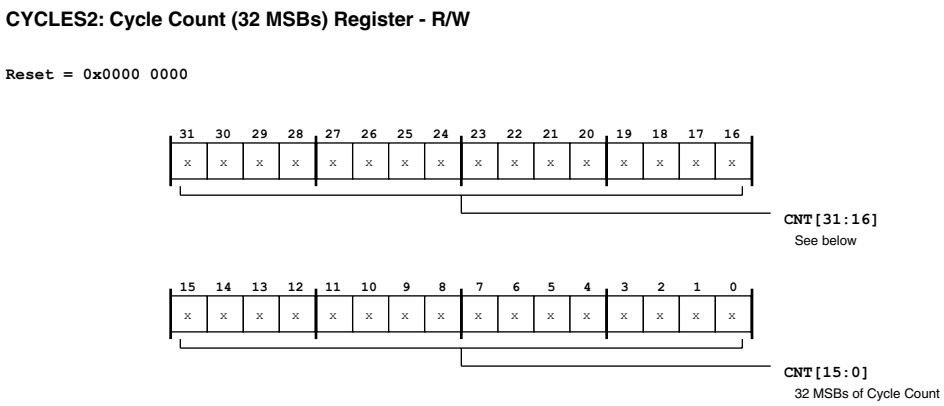


Figure 9-14: CYCLES2 Register Diagram

Table 9-25: CYCLES2 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	32 MSBs of Cycle Count. The CYCLES2.CNT bits hold the most significant 32 bits of the cycle count value.

10 Numeric Formats

ADSP-BF5xx Blackfin family processors support 8-, 16-, 32-, and 40-bit fixed-point data in hardware. Special features in the computation units allow support of other formats in software. This appendix describes various aspects of these data formats. It also describes how to implement a block floating-point format in software.

Unsigned or Signed: Two's-complement Format

Unsigned integer numbers are positive, and no sign information is contained in the bits. Therefore, the value of an unsigned integer is interpreted in the usual binary sense. The least significant words of multiple-precision numbers are treated as unsigned numbers.

Signed numbers supported by the ADSP-BF5xx Blackfin family are in two's-complement format. Signed-magnitude, one's-complement, binary-coded decimal (BCD) or excess-n formats are not supported.

Integer or Fractional Data Formats

The ADSP-BF5xx Blackfin family supports both fractional and integer data formats. In an integer, the radix point is assumed to lie to the right of the least significant bit (LSB), so that all magnitude bits have a weight of 1 or greater. This format is shown in the **Integer Format** figure. Note in two's-complement format, the sign bit has a negative weight.

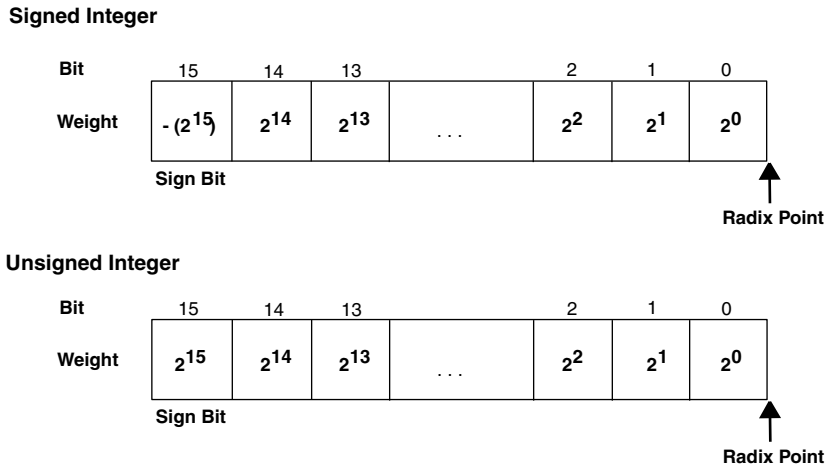


Figure 10-1: Integer Format

In a fractional format, the assumed radix point lies within the number, so that some or all of the magnitude bits have a weight of less than 1. In the format shown in the **Example of Fractional Format** figure, the assumed radix point lies to the left of the three LSBs, and the bits have the weights indicated.

The native formats for the Blackfin processor family are a signed fractional 1.M format and an unsigned fractional 0.N format, where N is the number of bits in the data word and M = N - 1.

The notation used to describe a format consists of two numbers separated by a period (.); the first number is the number of bits to the left of the radix point, the second is the number of bits to the right of the radix point. For example, 16.0 format is an integer format; all bits lie to the left of the radix point. The format in the **Example of Fractional Format** figure is 13.3.

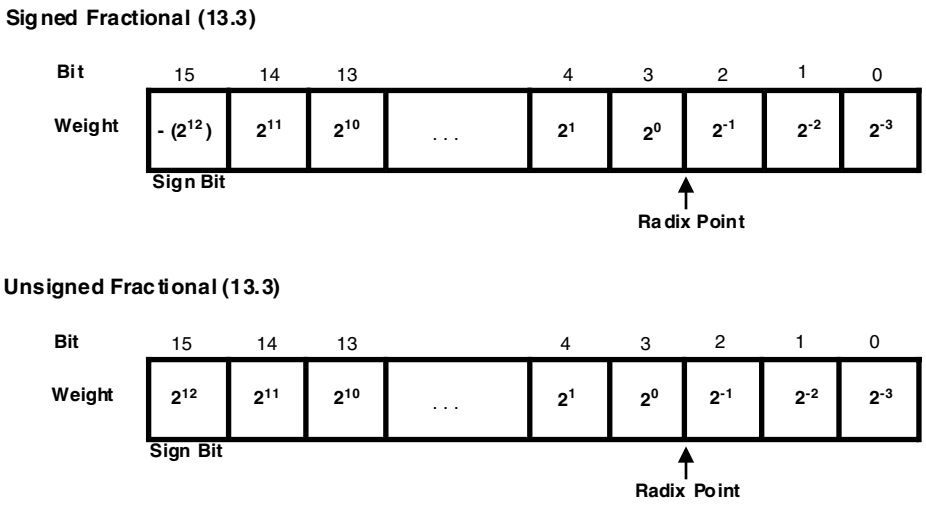


Figure 10-2: Example of Fractional Format

The **Fractional Formats and Their Ranges** table shows the ranges of signed numbers representable in the fractional formats that are possible with 16 bits.

Table 10-1: Fractional Formats and Their Ranges

Format	# of Integer Bits	# of Fractional Bits	Max Positive Value (0x7FFF) In Decimal	Max Negative Value (0x8000) In Decimal	Value of 1 LSB (0x0001) In Decimal
1.15	1	15	0.999969482421875	-1.0	0.000030517578125
2.14	2	14	1.999938964843750	-2.0	0.000061035156250
3.13	3	13	3.999877929687500	-4.0	0.000122070312500
4.12	4	12	7.999755859375000	-8.0	0.000244140625000
5.11	5	11	15.999511718750000	-16.0	0.000488281250000
6.10	6	10	31.999023437500000	-32.0	0.000976562500000
7.9	7	9	63.998046875000000	-64.0	0.001953125000000
8.8	8	8	127.996093750000000	-128.0	0.003906250000000

Table 10-1: Fractional Formats and Their Ranges (Continued)

Format	# of Integer Bits	# of Fractional Bits	Max Positive Value (0x7FFF) In Decimal	Max Negative Value (0x8000) In Decimal	Value of 1 LSB (0x0001) In Decimal
9.7	9	7	255.992187500000000	-256.0	0.007812500000000
10.6	10	6	511.984375000000000	-512.0	0.015625000000000
11.5	11	5	1023.968750000000000	-1024.0	0.031250000000000
12.4	12	4	2047.937500000000000	-2048.0	0.062500000000000
13.3	13	3	4095.875000000000000	-4096.0	0.125000000000000
14.2	14	2	8191.750000000000000	-8192.0	0.250000000000000
15.1	15	1	16383.500000000000000	-16384.0	0.500000000000000
16.0	16	0	32767.000000000000000	-32768.0	1.000000000000000

Binary Multiplication

In addition and subtraction, both operands must be in the same format (signed or unsigned, radix point in the same location), and the result format is the same as the input format. Addition and subtraction are performed the same way whether the inputs are signed or unsigned.

In multiplication, however, the inputs can have different formats, and the result depends on their formats. The ADSP-BF5xx Blackfin family assembly language allows you to specify whether the inputs are both signed, both unsigned, or one of each (mixed-mode). The location of the radix point in the result can be derived from its location in each of the inputs. This is shown in the **Format of Multiplier Result** figure. The product of two 16-bit numbers is a 32-bit number. If the inputs' formats are M.N and P.Q, the product has the format (M + P).(N + Q). For example, the product of two 13.3 numbers is a 26.6 number. The product of two 1.15 numbers is a 2.30 number.

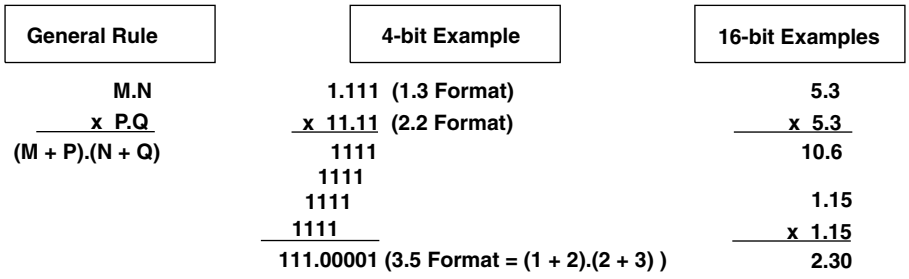


Figure 10-3: Format of Multiplier Result

Fractional Mode And Integer Mode

A product of 2 two's-complement numbers has two sign bits. Since one of these bits is redundant, you can shift the entire result left one bit. Additionally, if one of the inputs was a 1.15 number, the left shift causes

the result to have the same format as the other input (with 16 bits of additional precision). For example, multiplying a 1.15 number by a 5.11 number yields a 6.26 number. When shifted left one bit, the result is a 5.27 number, or a 5.11 number plus 16 LSBs.

The ADSP-BF5xx Blackfin family provides a means (a signed fractional mode) by which the multiplier result is always shifted left one bit before being written to the result register. This left shift eliminates the extra sign bit when both operands are signed, yielding a result that is correctly formatted.

When both operands are in 1.15 format, the result is 2.30 (30 fractional bits). A left shift causes the multiplier result to be 1.31 which can be rounded to 1.15. Thus, if you use a signed fractional data format, it is most convenient to use the 1.15 format.

Block Floating-Point Format

A block floating-point format enables a fixed-point processor to gain some of the increased dynamic range of a floating-point format without the overhead needed to do floating-point arithmetic. However, some additional programming is required to maintain a block floating-point format.

A floating-point number has an exponent that indicates the position of the radix point in the actual value. In block floating-point format, a set (block) of data values share a common exponent. A block of fixed-point values can be converted to block floating-point format by shifting each value left by the same amount and storing the shift value as the block exponent.

Typically, block floating-point format allows you to shift out non-significant MSBs (most significant bits), increasing the precision available in each value. Block floating-point format can also be used to eliminate the possibility of a data value overflowing. See the **Data With Guard Bits** figure. Each of the three data samples shown has at least two non-significant, redundant sign bits. Each data value can grow by these two bits (two orders of magnitude) before overflowing. These bits are called guard bits.

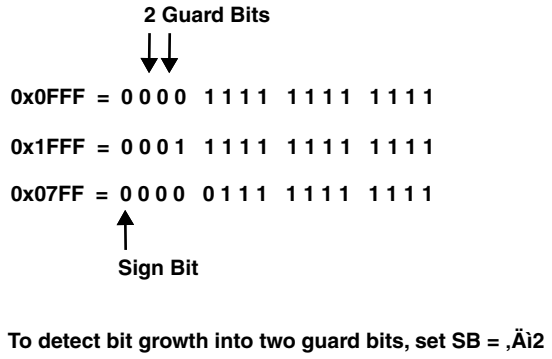


Figure 10-4: Data With Guard Bits

If it is known that a process will not cause any value to grow by more than the two guard bits, then the process can be run without loss of data. Later, however, the block must be adjusted to replace the guard bits before the next process.

The **Block Floating-point Adjustment** figure shows the data after processing but before adjustment. The block floating-point adjustment is performed as follows.

- Assume the output of the `SIGNBITS` instruction is `SB` and `SB` is used as an argument in the `EXPADJ` instruction. Initially, the value of `SB` is +2, corresponding to the two guard bits. During processing, each resulting data value is inspected by the `EXPADJ` instruction, which counts the number of redundant sign bits and adjusts `SB` if the number of redundant sign bits is less than two. In this example, `SB` = +1 after processing, indicating the block of data must be shifted right one bit to maintain the two guard bits.
- If `SB` were 0 after processing, the block would have to be shifted two bits right. In either case, the block exponent is updated to reflect the shift.

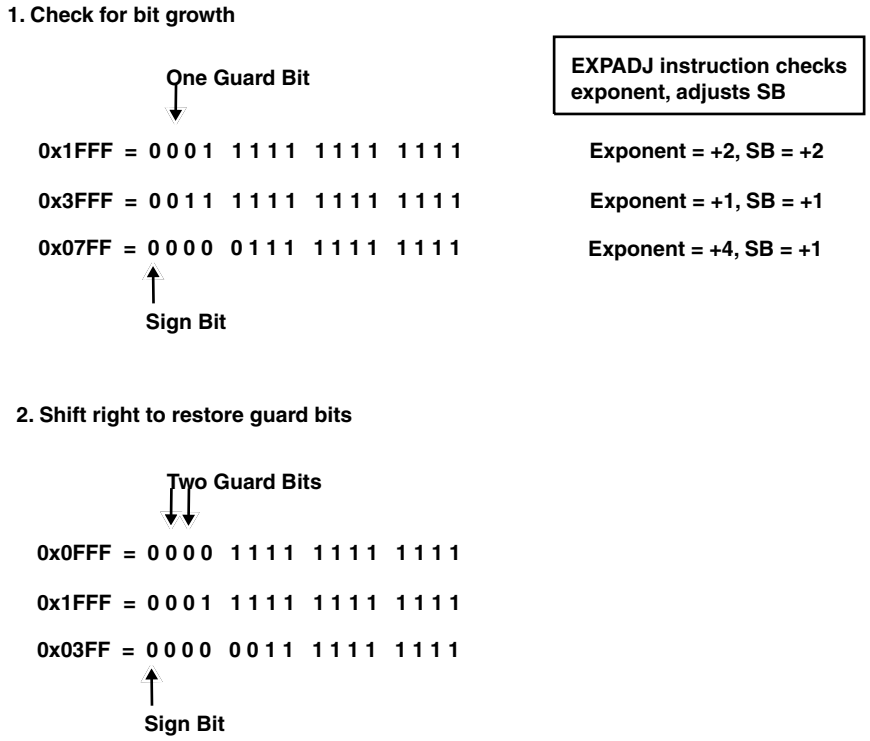


Figure 10-5: Block Floating-point Adjustment

I Index

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z