BLACKfin™ DSP

Instruction Set Reference

Preliminary Edition, November 2001

Part Number
82-000410-14

Analog Devices, Inc.
DSP and Systems Products Group
Three Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

Copyright Information

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

Development Information

Blackfin DSPs are based on the Micro Signal Architecture, jointly developed by Analog Devices, Inc. and Intel Corporation.

Contents

FIGURES

TABLES

INTRODUCTION 1

The ***Blackfin DSP Instruction Set Reference*** provides details about the assembly language instructions used by the BlackfinDSP core developed jointly by Analog Devices, Inc. and Intel Corporation. This section points out some of the conventions used in this document.

1.1 Manual Organization

The instructions are grouped according to their functions. Within groupings, the instructions are generally arranged alphabetically unless a functional relationship makes another order clearer for the programmer. One such example of non-alphabetic ordering is the Load / Store section where the Load Pointer Register appears before a pile of seven Load Data Register derivations. The instructions are listed at the beginning of each chapter in the order they appear.

The instruction groups, or chapters, are arranged according to complexity, beginning with the basic Program Flow Control and Load / Store chapters and progressing to Video Pixel Operations and Vector Operations.

1.2 Syntax Conventions

The Blackfin instruction set supports several syntactic conventions that appear throughout this document. Those conventions are given below.

1.2.1 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.

1.2.2 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 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 multi-character special symbols like "+=", "/*", or "||".

1.2.3 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 operation will consist of more than one operation. There are two cases:

- 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++];

1.2.4 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 ended 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.

1.3 Notation Conventions

This manual and the assembler use the following conventions:

- Register names are alphabetic, 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.
- Some operations require a register pair. Register pairs are always Data Registers and are denoted using a colon, i.e., R3:2. The larger number must is written first. Note: The hardware supports only odd-even pairs. i.e., R7:6, R5:4, R3:2, and R1:0.
- Some instructions require a group of adjacent registers. Adjacent registers are denoted by the range enclosed in brackets, i.e., R[7:3]. Again, the larger number appears first.
- Portions of a particular register may be individually specified. This is written 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 elipsis marks ("..."). For example, "R0, ..., R7" means that any one of the eight Data Registers can be used.
- Immediate values are designated as "imm" with the following modifiers:
  - — "imm" indicates a signed value; for example imm7.
  - — the "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.

1.4 Behavior Conventions

1.4.1 Accumulator Saturation

All operations that produce a result in an Accumulator saturate to a 40 bit quantity unless noted otherwise. See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

1.5 Glossary

The following terms appear throughout this document. Without trying to explain the Blackfin, here are the terms used with their definitions. See the ***ADSP-21535 Blackfin DSP Hardware Reference*** for details about the architecture.

1.5.1 Register Names

The architecture includes the following registers:

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 as four registers – one 32-bit register (designated as A1.W or A0.W), two 16-bit registers similar to Data registers (designated as A1.H, A1.L, A0.H, or A0.L) and one 8-bit register (designated A1.X or A0.X) for the bits that extend beyond bit 31.
Data Registers	The set of 32-bit registers R0, R1, …, R6, R7 that normally contain data for manipulation. Abbreviated D-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, …, P4, P5, including SP and FP that normally contain byte addresses of data structures. Accessed only as a 32-bit register. Abbreviated P-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 I-register or Ireg.

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 as four registers – one 32-bit register (designated as A1.W or A0.W), two 16-bit registers similar to Data registers (designated as A1.H, A1.L, A0.H, or A0.L) and one 8-bit register (designated A1.X or A0.X) for the bits that extend beyond bit 31.
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.
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.

1.5.2 Functional Units

The architecture includes two processor sections:

Table 1-2. Processor Sections

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 two sections (MAC0 and MAC1) each associated with an Accumulator (A0 and A1, respectively).

1.5.3 Arithmetic Status Flags

The Micro Signal Architecture (MSA) includes 12 arithmetic status flags that indicate specific results of a prior operation. These flags reside in the Arithmetic Status (ASTAT) Register. A summary of the flags appears below. All flags are active high. Instructions regarding P-registers, I-registers, L-registers, M-registers, or B-registers do not affect flags.

See the ***ADSP-21535 Blackfin DSP Hardware Reference*** for details.

Table 1-3. Arithmetic Status Flag Summary

Flag	Description
AC0	Carry (ALU0)
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

Table 1-3. Arithmetic Status Flag Summary

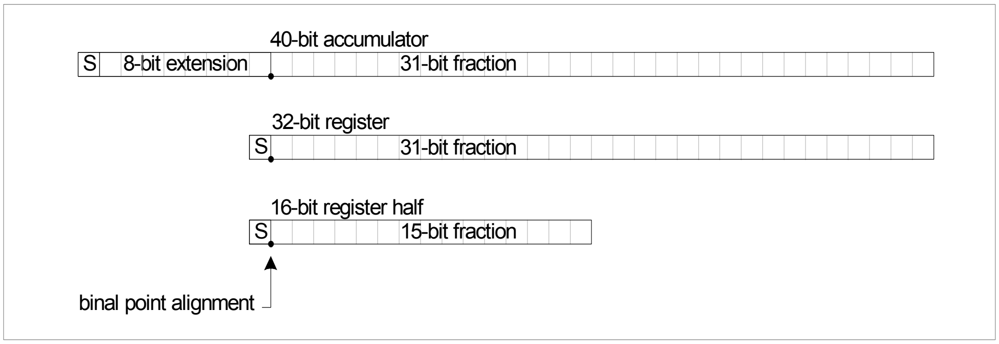
Flag	Description
AVS1	Accumulator 1 Sticky Overflow; set when AV1 is set, but remains set until explicitly cleared by user code
AZ	Zero
CC	Control Code bit; multi-purpose flag set, cleared and tested by specific instructions
V	Overflow for Data Register results
VS	Sticky Overflow for Data Register results; set when V is set, but remains set until explicitly cleared by user code

1.5.4 Fractional Convention

Fractional numbers include sub-integer 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*.

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. For 40-bit registers, data is assumed to be in 9.31 fraction notation, where there are 31 fractional bits, 8 extension bits, and one sign bit. For 32-bit registers, data is represented in 1.31 fraction notation, with one sign bit and 31 fractional bits. For 16-bit registers, data is represented in 1.15 fractional notation. This processor does not represent fractional values in 8-bit registers.

Figure 1-1. Conventional Placement of Binal Point Within 40-, 32-, and 16-Bit Data



1.5.5 Saturation

When the result of a 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*, saturation limits the result to the maximum *positive* value for the size register being used, instead.

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

The overflow arithmetic flag is never set by an operation that enforces saturation.

The maximum positive value in a 16-bit register is 0x7FFF . The maximum negative value is 0x8000. For a signed 2'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 2'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 2's complement 9.31 fractional notation, the range of values that can be represented is -256 through $(256-2^{-31})$.

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 MSA implements 40-bit saturation for all arithmetic operations that write an 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 MSA 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. 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 flags, as noted in the instruction descriptions. The arithmetic flags reside in the Arithmetic Status (ASTAT) register. See the ***ADSP-21535 Blackfin DSP Hardware Reference*** for details about the ASTAT register.

1.5.6 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, 2's complement fraction 0.25 (binary 0.01) to the nearest 2-bit 2'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*.

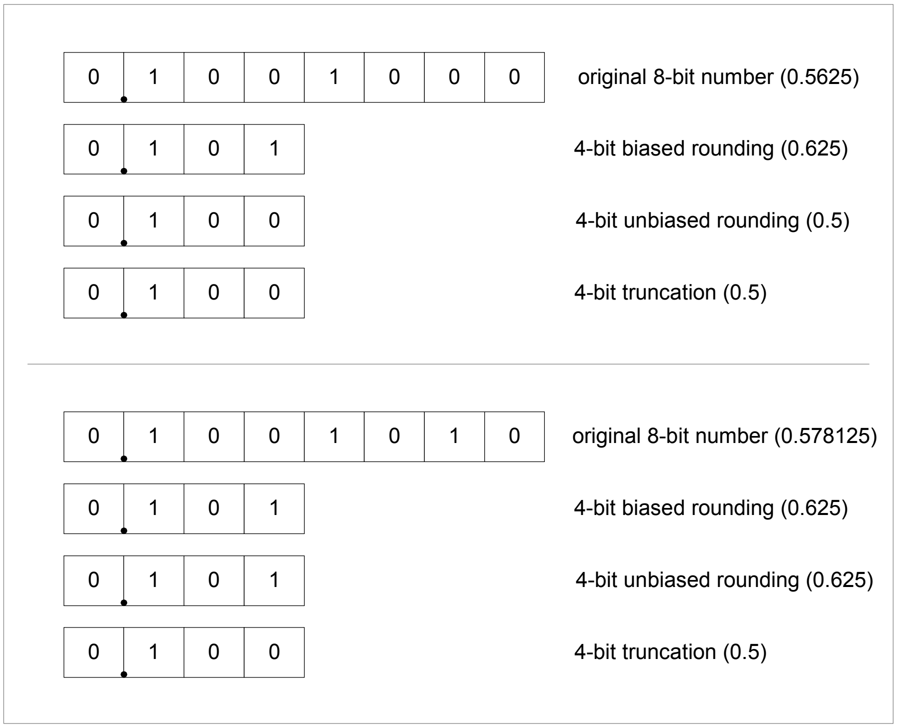
The **convergent rounding** method also returns the closest number to the original. However, in cases where the original number lies exactly halfway between two numbers, this method returns the nearest even number, the one containing an LSB of 0. So for the example above, the result would be 0.0, since 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. See the ***ADSP-21535 Blackfin DSP Hardware Reference*** for details about the ASTAT register.

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 figures below show other examples of rounding and truncation methods.

Figure 1-2. Two Examples Showing an 8-Bit Number Reduced to 4 Bits of Precision


0 1 0 0 1 0 0 0 original 8-bit number (0.5625)
0 1 0 1 4-bit biased rounding (0.625)
0 1 0 0 4-bit unbiased rounding (0.5)
0 1 0 0 4-bit truncation (0.5)
0 1 0 0 1 0 1 0 original 8-bit number (0.578125)
0 1 0 1 4-bit biased rounding (0.625)
0 1 0 1 4-bit unbiased rounding (0.625)
0 1 0 0 4-bit truncation (0.5)

1.6 Related References

The ***ADSP-21535 Blackfin DSP Hardware Reference*** describes the Blackfin architecture, register set, and behavior.

1.7 Document Errata Sightings

This document contains errors, but the authors do not know about them, yet. Errata sightings are expected and encouraged. Please send comments, criticisms, and encouragements to Mike.J.Pulley@intel.com with “MSA ISR Feedback: <your descriptive title, here>” (without the quotes) in the subject line. All valid errata sightings are appreciated and will be incorporated in future releases of this document.

PROGRAM FLOW CONTROL 2

Instruction Summary

This chapter discusses the instructions that control program flow. 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.

2.1 Jump

2.1.1 General Form

JUMP

JUMP.S

JUMP.L

2.1.2 Syntax

JUMP (Preg) ;	/* indirect to an absolute (not PC-relative) address (a) */
JUMP (PC + Preg) ;	// PC relative, indexed (a)
JUMP pcrelm2 ;	/* PC relative, immediate [1] (a) or (b), see Section 2.1.5, "Functional Description" */
JUMP.S pcrel13m2 ;	// PC relative, immediate, short (a)
JUMP.L pcrel25m2 ;	// PC relative, immediate, long (b)

2.1.3 Syntax Terminology

Preg: P0, ..., P5, SP, FP

pcrelm2: undetermined 25-bit or smaller signed, even relative offset, with a range of -16,777,216 through 16,777,214 bytes (0xFF00 0000 to 0x00FF FFFE)

pcrel13m2: 13-bit signed, even relative offset, with a range of -4096 through 4094 bytes (0xF000 to 0x0FFE)

pcrel25m2: 25-bit signed, even relative offset, with a range of -16,777,216 through 16,777,214 bytes (0xFF00 0000 to 0x00FF FFFE)

2.1.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

2.1.5 Functional Description

The Jump instruction forces a new value into the Program Counter (PC) to change program flow.

In the Indirect and Indexed versions of the instruction, the value in Preg must be an even number (bit0=0) to maintain 16-bit address alignment. Otherwise, an odd offset in Preg causes the processor to invoke an alignment exception.

[1] This instruction can be used in assembly-level programs when the final distance to the target is unknown at coding time. The assembler substitutes the opcode for JUMP.S or JUMP.L depending on the final target. Disassembled code shows the mnemonic JUMP.S or JUMP.L.

2.1.6 Flags Affected

None

2.1.7 Required Mode

User & Supervisor

2.1.8 Parallel Issue

The Jump instruction cannot be issued in parallel with other instructions.

2.1.9 Example

```
jump ( p5 ) ;
jump ( pc + p2 ) ;
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 */
jump get_new_sample ;             // assembler resolved target, abstract offsets
```

2.1.10 Also See

Branch, Call

2.1.11 Special Applications

2.2 Conditional Jump

2.2.1 General Form

IF CC JUMP

IF !CC JUMP

2.2.2 Syntax

```
IF CC JUMP pcrell1m2 ;              // Branch if CC=1, branch predicted as not taken [2] (a)
IF CC JUMP pcrel11m2 (bp) ;         // Branch if CC=1, branch predicted as taken [2] (a)
IF !CC JUMP pcrel11m2 ;             // Branch if CC=0, branch predicted as not taken [3] (a)
IF !CC JUMP pcrel11m2 (bp) ;        // Branch if CC=0, branch predicted as taken [3] (a)
```

2.2.3 Syntax Terminology

pcrel11m2: 11-bit signed even relative offset, with a range of -1024 through 1022 bytes (0xFC00 to 0x03FE). This value can optionally be replaced with an address label that is evaluated and replaced during linking.

2.2.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

2.2.5 Functional Description

The 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.

The range of valid offset values is –1024 through 1022.

2.2.6 Option

The Branch Prediction appendix (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.

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).

[2] CC bit = 1 causes a branch to an address, computed by adding the signed, even offset to the current PC value.

[3] CC bit = 0 causes a branch to an address, computed by adding the signed, even relative offset to the current PC value.

2.2.7 Flags Affected

None

2.2.8 Required Mode

User & Supervisor

2.2.9 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

2.2.10 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 */
```

2.2.11 Also See

Jump, Call

2.2.12 Special Applications

2.3 Call

2.3.1 General Form

CALL

2.3.2 Syntax

CALL (Preg) ;	/* indirect to an absolute (not PC-relative) address (a) */
CALL (PC + Preg) ;	// PC-relative, indexed (a)
CALL pcrel25m2 ;	// PC-relative, immediate (b)

2.3.3 Syntax Terminology

Preg: P0, ..., P5 (SP and FP are not allowed as the source register for this instruction.)

pcrel25m2: 25-bit signed, even, PC-relative offset; can be specified as a symbolic address label, with a range of -16,777,216 through 16,777,214 (0xFF00 0000 to 0x00FF FFFE) bytes.

2.3.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

2.3.5 Functional Description

The Call instruction calls a subroutine from an address that a P-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 value in the Preg must be an even value to maintain 16-bit alignment.

2.3.6 Flags Affected

None

2.3.7 Required Mode

User & Supervisor

2.3.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

2.3.9 Example

```
call ( p5 ) ;
call ( pc + p2 ) ;
call 0x123456 ;
call get_next_sample ;
```

2.3.10 Also See

Return, Jump, Conditional Jump

2.3.11 Special Applications

2.4 Return

2.4.1 General Form

RTS, RTI, RTX, RTN, RTE

2.4.2 Syntax

```
RTS ;                                   // Return from Subroutine (a)
RTI ;                                   // Return from Interrupt (a)
RTX ;                                   // Return from exception (a)
RTN ;                                   // Return from NMI (a)
RTE ;                                   // Return from Emulation (a)
```

2.4.3 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

2.4.4 Functional Description

The Return instruction forces a return from a subroutine, maskable or NMI interrupt routine, exception routine, or emulation routine (see Table 2-1).

Table 2-1. Types of Return Instruction

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 loading the value of the RETN register into the PC.
RTE	Forces a return from an emulation routine and emulation mode by loading 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.

2.4.5 Flags Affected

None

2.4.6 Required Mode

Table 2-2 identifies the modes required by the Return instruction.

Table 2-2. Required Mode for the Return Instruction

Mnemonic	Required Mode
RTS	User & 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 Supervisor mode produces an exception.

2.4.7 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

2.4.8 Example

```
rts ;
rti ;
rtx ;
rtn ;
rte ;
```

2.4.9 Also See

Call, Push, Pop

2.4.10 Special Applications

2.5 Zero-Overhead Loop Setup

2.5.1 General Form

First Form

```
LSETUP ( Begin_Loop , End_Loop ) Loop_Counter
```

Second Form

```
LOOP loop_name loop_counter
LOOP_BEGIN loop_name
LOOP_END loop_name
```

2.5.2 Syntax

For Loop0

```
LSETUP ( pcrel5m2 , lppcrel11m2 ) LC0 ;              // (b)
LSETUP ( pcrel5m2 , lppcrel11m2 ) LC0 = Preg ;       // autoinitialize LC0 (b)
LSETUP ( pcrel5m2 , lppcrel11m2 ) LC0 = Preg >> 1 ;// autoinitialize LC0 (b)
LOOP loop_name LC0 ;                                 // (b)
LOOP loop_name LC0 = Preg ;                          // autoinitialize LC0 (b)
LOOP loop_name LC0 = Preg >> 1 ;                     // autoinitialize LC0 (b)
```

For Loop1

```
LSETUP ( pcrel5m2 , lppcrel11m2 ) LC1 ;              // (b)
LSETUP ( pcrel5m2 , lppcrel11m2 ) LC1 = Preg ;       // autoinitialize LC1 (b)
LSETUP ( pcrel5m2 , lppcrel11m2 ) LC1 = Preg >> 1 ;// autoinitialize LC1 (b)
LOOP loop_name LC1 ;                                 // (b)
LOOP loop_name LC1 = Preg ;                          // autoinitialize LC1 (b)
LOOP loop_name LC1 = Preg >> 1 ;                     // autoinitialize LC1 (b)

LOOP_BEGIN loop_name ;                               // define the first instruction of the loop (b)
LOOP_END loop_name ;                                 // define the last instruction of the loop (b)
```

2.5.3 Syntax Terminology

Preg: P0, ..., P5 (SP and FP are not allowed as the source register for this instruction.)

pcrel5m2: 5-bit unsigned, even, PC-relative offset; can be replaced by a symbolic label. The range is 4 to 30, or 2^5 -2.

lppcrel11m2: 11-bit unsigned, even, PC-relative offset for a loop; can be replaced by a symbolic label. The range is 4 to 2046 (0x0004 to 0x07FE), or 2^{11} -2.

loop_name: a symbolic address label

2.5.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

2.5.5 Functional 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 penalty for decrementing a counter, evaluating a loop condition, then calculating and branching to a new target address.

The architecture includes two sets of three registers each to support two independent, nestable loops. The registers are Loop_Top (LTn), Loop_Bottom (LBn) and Loop_Count (LCn). Consequently, LT0, LB0, and LC0 describe Loop0, and LT1, LB1, and LC1 describe Loop1.

The LSETUP and LOOP instructions are a convenient way to intialize all three registers in a single instruction. The size of the LSETUP and LOOP instructions can only support just so many 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 LSETUP or LOOP syntax. Thus, a single loop can span the entire 4GB of memory space.

The LSETUP instruction accepts an optional initialization value from a P-register or P-register divided by 2.

An alternative syntax to accomplish the same result is the LOOP, LOOP_BEGIN, LOOP_END instruction sequence. This syntax contains the same information as the LSETUP syntax, but in a more readable, user-friendly format.

If LCn is non-zero when the fetch address equals LBn, the processor decrements LCn and places the address in LTn into the PC. The loop always executes once through since the Loop_Count is evaluated at the end of the loop.

A value of 0 (zero) in the Loop_Count disables the hardware loop mechanism, 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, 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 LTn, 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 LBn, 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, as usual.

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.

The last instruction of the LSETUP loop must not be a branch instruction. As long as the hardware loop is active (that is., the Loop_Count is non-zero), a branch instruction at the End_Loop address produces undefined execution. If a branch instruction comes last in the LSETUP loop, no exception is generated. Branch instructions that are located anywhere else in the defined loop execute normally.

Also, the last instruction in the LSETUP loop must not modify the registers that define the currently active loop (LCn, LTn, or LBn). User modifications to those registers while the hardware accesses them produces undefined execution. Software can legally modify the loop counter at any other location in the loop.

2.5.6 Flags Affected

None

2.5.7 Required Mode

User & Supervisor

2.5.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

2.5.9 Example

```
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 ;
```

```
loop DoItSome LC0 ;                 /* define loop ‘DoItSome’ with Loop
                                     Counter 0 */
loop_begin DoItSome ;               /* place before the first instruction in the
                                     loop */

loop_end DoItSome ;                 /* place after the last instruction in the
                                     loop */


loop MyLoop LC1 ;                   /* define loop ‘MyLoop’ with Loop
                                     Counter 1 */
loop_begin MyLoop ;                 /* place before the first instruction in the
                                     loop */

loop_end MyLoop ;                   /* place after the last instruction in the
                                     loop */
```

2.5.10 Also See

Conditional Jump, Jump

2.5.11 Special Applications

LOAD / STORE 3

Instruction Summary

This chapter discusses the load / store instructions. Users can take advantage of these instructions to load and store immediate values, pointer registers, data registers or data register halves, and half-words (zero- or sign-extended).

3.1 Load Immediate

3.1.1 General Form

register = constant

A1 = A0 = 0

3.1.2 Syntax

NOT EXTENDED

```
reg_lo = uimm16 ;          // 16-bit value into low-half data or address register (b)
reg_hi = uimm16 ;          // 16-bit value into high-half data or address register (b)
```

ZERO-EXTENDED

```
reg = uimm16 (Z) ;         // 16-bit value, zero-extended, into data or address register (b)
A0 = 0 ;                   // Clear A0 register (b)
A1 = 0 ;                   // Clear A1 register (b)
A1 = A0 = 0 ;              // Clear A1 and A0 register at once (b)
```

SIGN-EXTENDED

```
Dreg = imm7 (X) ;          // 7-bit value, sign extended, into Dreg (a)
Preg = imm7 (X) ;          // 7-bit value, sign extended, into Preg (a)
reg = imm16 (X) ;          // 16-bit value, sign extended, into data or address register (b)
```

3.1.3 Syntax Terminology

Dreg: R0, ..., R7

Preg: P0, ..., P5, SP, FP

reg_lo: R0.L, ..., R7.L, P0.L, ..., P5.L, SP.L, FP.L, I0.L, ..., I3.L, M0.L, ..., M3.L, B0.L, ..., B3.L, L0.L, ..., L3.L

reg_hi: R0.H, ..., R7.H, P0.H, ..., P5.H, SP.H, FP.H, I0.H, ..., I3.H, M0.H, ..., M3.H, B0.H, ..., B3.H, L0.H, ..., L3.H

reg: R0, ..., R7, P0, ..., P5P0, ..., P5, SP, FP, I0, ..., I3, M0, ..., M3, B0, ..., B3, L0, ..., L3

imm7: 7-bit signed field, with a range of -64 through 63

imm16: 16-bit signed field, with a range of -32,768 through 32,767 (0x800 through 0x7FFF)

uimm16: 16-bit unsigned field, with a range of 0 through 65,535 (0x0000 through 0xFFFF)

3.1.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

3.1.5 Functional Description

The Load Immediate instruction loads immediate values, or explicit constants, into registers.

The instruction loads a 7-bit or 16-bit quantity, depending on the size of the immediate data. The range of constants that can be loaded is 0x8000 through 0x7FFF, equivalent to –32768 through +32767.

The only values that can be immediately loaded into 40-bit Accumulator registers are zeroes.

Sixteen-bit half-words can be loaded into either the high half or low half of a register. The load operation leaves the unspecified half of the register intact.

The zero-extended versions fill the upper bits of the destination register with zeros. The sign-extended versions fill the upper bits with the sign of the constant value.

3.1.6 Flags Affected

None

3.1.7 Required Mode

User & Supervisor

3.1.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

3.1.9 Example

```
r7 = 63 (z) ;
p3 = 12 (z) ;
r0 = -344 (x) ;
r7 = 436 (z) ;
m2 = 0x89ab (z) ;
p1 = 0x1234 (z) ;
m3 = 0x3456 (x) ;
l3.h = 0xbcde ;
a0 = 0 ;
a1 = 0 ;
a1 = a0 = 0 ;
```

3.1.10 Also See

Load Data Register, Load Pointer Register

3.1.11 Special Applications

Use the Load Immediate instruction to initialize registers.

3.2 Load Pointer Register

3.2.1 General Form

P-register = [indirect_address]

3.2.2 Syntax

```
Preg = [ Preg ] ;                    // indirect (a)
Preg = [ Preg ++ ] ;                 // indirect, post-increment (a)
Preg = [ Preg -- ] ;                 // indirect, post-decrement (a)
Preg = [ Preg + uimm6m4 ] ;          // indexed with small offset (a)
Preg = [ Preg + uimm17m4 ] ;         // indexed with large offset (b)
Preg = [ Preg - uimm17m4 ] ;         // indexed with large offset (b)
Preg = [ FP - uimm7m4 ] ;            // indexed FP-relative (a)
```

3.2.3 Syntax Terminology

Preg: P0, ..., P5, SP, FP

uimm6m4: 6-bit unsigned field that must be a multiple of 4, with a range of 0 through 60 bytes

uimm7m4: 7-bit unsigned field that must be a multiple of 4, with a range of 4 through 128 bytes

uimm17m4: 17-bit unsigned field that must be a multiple of 4, with a range of 0 through 131,068 bytes (0x0000 0000 through 0x0001 FFFC)

3.2.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

3.2.5 Functional Description

The Load Pointer Register instruction loads a 32-bit P-register with a 32-bit word from an address specified by a P-register.

The indirect address and offset must yield an even multiple of 4 to maintain 4-byte word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

3.2.6 Options

The Load Pointer Register instruction supports the following options:

- Post-increment the source pointer by 4 bytes
- Post-decrement the source pointer by 4 bytes

- Offset the source pointer with a small (6-bit), word-aligned (multiple of 4), unsigned constant.
- Offset the source pointer with a large (18-bit), word-aligned (multiple of 4), signed constant.
- Frame Pointer (FP) relative and offset with a 7-bit, word-aligned (multiple of 4), negative constant.

The indexed FP-relative form is typically used to access local variables in a subroutine or function. Positive offsets relative to FP (such as is useful to access arguments from a called function) can be accomplished using one of the other versions of this instruction. Preg includes the Frame Pointer and Stack Pointer.

Auto-increment or auto-decrement pointer registers cannot also be the destination of a Load instruction. For example, sp=[sp++] is not a valid instruction because it prescribes two competing values for the Stack Pointer -- (a) the data returned from memory and (b) post-incremented SP++. Similarly, P0=[P0++] and P1=[P1++], etc. are invalid. Such an instruction causes an undefined instruction exception.

3.2.7 Flags Affected

None

3.2.8 Required Mode

User & Supervisor

3.2.9 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

3.2.10 Example

```
p3 = [ p2 ] ;
p5 = [ p0 ++ ] ;
p2 = [ sp -- ] ;
p3 = [ p2 + 8 ] ;
p0 = [ p2 + 0x4008 ] ;
p1 = [ fp - 16 ] ;
```

3.2.11 Also See

Load Immediate, Pop, Pop Multiple

3.2.12 Special Applications

3.3 Load Data Register

3.3.1 General Form

D-register = [indirect_address]

3.3.2 Syntax

```
Dreg = [ Preg ] ;                  // indirect (a)
Dreg = [ Preg ++ ] ;               // indirect, post-increment (a)
Dreg = [ Preg -- ] ;               // indirect, post-decrement (a)
Dreg = [ Preg + uimm6m4 ] ;        // indexed with small offset (a)
Dreg = [ Preg + uimm17m4 ] ;       // indexed with large offset (b)
Dreg = [ Preg - uimm17m4 ] ;       // indexed with large offset (b)
Dreg = [ Preg ++ Preg ] ;          // indirect, post-increment index [1] (a)
Dreg = [ FP - uimm7m4 ] ;          // indexed FP-relative (a)
Dreg = [ Ireg ] ;                  // indirect (a)
Dreg = [ Ireg ++ ] ;               // indirect, post-increment (a)
Dreg = [ Ireg -- ] ;               // indirect, post-decrement (a)
Dreg = [ Ireg ++ Mreg ] ;          // indirect, post-increment index [1] (a)
```

3.3.3 Syntax Terminology

Dreg: R0, ..., R7

Preg: P0, ..., P5, SP, FP

Ireg: I0, ..., I3

Mreg: M0, ..., M3

uimm6m4: 6-bit unsigned field that must be a multiple of 4, with a range of 0 through 60 bytes

uimm7m4: 7-bit unsigned field that must be a multiple of 4, with a range of 4 through 128 bytes

uimm17m4: 17-bit unsigned field that must be a multiple of 4, with a range of 0 through 131,068 bytes (0x0000 0000 through 0x0001 FFFC)

3.3.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

[1] See “Indirect and Post-Increment Index Addressing” on page 3-7.

3.3.5 Functional Description

The Load Data Register instruction loads a 32-bit word into a 32-bit D-register from a memory location. The source pointer register can be a P-register, I-register, or the Frame Pointer.

The indirect address and offset must yield an even multiple of 4 to maintain 4-byte word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

3.3.6 Options

The Load Data Register instruction supports the following options:

- Post-increment the source pointer by 4 bytes to maintain word alignment
- Post-decrement the source pointer by 4 bytes to maintain word alignment
- Offset the source pointer with a small (6-bit), word-aligned (multiple of 4), unsigned constant.
- Offset the source pointer with a large (18-bit), word-aligned (multiple of 4), signed constant.
- Frame Pointer (FP) relative and offset with a 7-bit, word-aligned (multiple of 4), negative constant.

The indexed FP-relative form is typically used to access local variables in a subroutine or function. Positive offsets relative to FP (such as is useful to access arguments from a called function) can be accomplished using one of the other versions of this instruction. Preg includes the Frame Pointer and Stack Pointer.

3.3.7 Indirect and Post-Increment Index Addressing

The syntax of the form Dest = [Src_1 ++ Src_2] is indirect and 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 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 P-registers for the input operands. If a common Preg is used for the inputs, the auto-increment feature does not work.

3.3.8 Flags Affected

None

3.3.9 Required Mode

User & Supervisor

3.3.10 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

3.3.11 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
```

3.3.12 Also See

Load Immediate

3.3.13 Special Applications

3.4 Load Half-Word – Zero-Extended

3.4.1 General Form

D-register = W [indirect_address] (Z)

3.4.2 Syntax

Dreg = W [Preg] (Z);	// indirect (a)
Dreg = W [Preg ++] (Z);	// indirect, post-increment (a)
Dreg = W [Preg --] (Z);	// indirect, post-decrement (a)
Dreg = W [Preg + uimm5m2] (Z);	// indexed with small offset (a)
Dreg = W [Preg + uimm16m2] (Z);	// indexed with large offset (b)
Dreg = W [Preg - uimm16m2] (Z);	// indexed with large offset (b)
Dreg = W [Preg ++ Preg] (Z);	// indirect, post-increment index [2] (a)

3.4.3 Syntax Terminology

Dreg: R0, ..., R7

Preg: P0, ..., P5, SP, FP

uimm5m2: 5-bit unsigned field that must be a multiple of 2, with a range of 0 through 30 bytes

uimm16m2: 16-bit unsigned field that must be a multiple of 2, with a range of 0 through 65,534 bytes (0x0000 through 0xFFFC)

3.4.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

3.4.5 Functional Description

The Load Half-Word – Zero-Extended instruction loads 16 bits from a memory location into the lower half of a 32-bit data register. The instruction zero-extends the upper half of the register. The pointer register is a P-register.

The indirect address and offset must yield an even numbered address to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

[2] See “Indirect and Post-Increment Index Addressing” on page 3-10.

3.4.6 Options

The Load Half-Word – Zero-Extended instruction supports the following options:

- Post-increment the source pointer by 2 bytes
- Post-decrement the source pointer by 2 bytes
- Offset the source pointer with a small (5-bit), half-word-aligned (even), unsigned constant.
- Offset the source pointer with a large (17-bit), half-word-aligned (even), signed constant.

3.4.7 Indirect and Post-Increment Index Addressing

The syntax of the form Dest = W [Src_1 ++ Src_2] is indirect and 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 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 P-registers for the input operands. If a common Preg 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).

3.4.8 Flags Affected

None

3.4.9 Required Mode

User & Supervisor

3.4.10 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

3.4.11 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);
```

3.4.12 Also See

Load Half-Word Sign-Extended, Load Low Data Register Half, Load High Data Register Half, Load Data Register

3.4.13 Special Applications

To read consecutive, aligned 16-bit values for high-performance DSP operations, use the Load Data Register instructions instead of these half-word instructions. The half-word load instructions use only half the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.

3.5 Load Half-Word – Sign-Extended

3.5.1 General Form

D-register = W [indirect_address]

3.5.2 Syntax

```
Dreg = W [ Preg ] (X);                    // indirect (a)
Dreg = W [ Preg ++ ] (X);                 // indirect, post-increment (a)
Dreg = W [ Preg -- ] (X);                 // indirect, post-decrement (a)
Dreg = W [ Preg + uimm5m2 ] (X);          // indexed with small offset (a)
Dreg = W [ Preg + uimm16m2 ] (X);         // indexed with large offset (b)
Dreg = W [ Preg - uimm16m2 ] (X);         // indexed with large offset (b)
Dreg = W [ Preg ++ Preg ] (X);            // indirect, post-increment index [3] (a)
```

3.5.3 Syntax Terminology

Dreg: R0, ..., R7

Preg: P0, ..., P5, SP, FP

uimm5m2: 5-bit unsigned field that must be a multiple of 2, with a range of 0 through 30 bytes

uimm16m2: 16-bit unsigned field that must be a multiple of 2, with a range of -0 through 65,534 bytes (0x0000 through 0xFFFE)

3.5.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

3.5.5 Functional Description

The Load Half-Word – Sign-Extended instruction loads 16 bits sign-extended from a memory location into a 32-bit data register. The pointer register is a P-register. The MSB of the number loaded is replicated in the whole upper-half word of the destination D-register.

The indirect address and offset must yield an even numbered address to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

[3] See “Indirect and Post-Increment Index Addressing” on page 3-13.

3.5.6 Options

Optionally, (X) can be included in the instruction syntax to make it clearer that the load is sign extended. This optional flag does not affect the instruction encoding, function, or performance.

The Load Half-Word – Sign-Extended instruction supports the following options:

- Post-increment the source pointer by 2 bytes
- Post-decrement the source pointer by 2 bytes
- Offset the source pointer with a small (5-bit), half-word-aligned (even), unsigned constant.
- Offset the source pointer with a large (17-bit), half-word-aligned (even), signed constant.

3.5.7 Indirect and Post-Increment Index Addressing

The syntax of the form Dest = W [Src_1 ++ Src_2] (X) is indirect and 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 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 P-registers for the input operands. If a common Preg is used for the inputs, the instruction functions as a simple, non-incrementing load. For example, r0 = W[p2++p2] functions as r0 = W[p2].

3.5.8 Flags Affected

None

3.5.9 Required Mode

User & Supervisor

3.5.10 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

3.5.11 Example

```
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);
```

3.5.12 Also See

Load Half-Word – Zero Extended, Load Low Data Register Half, Load High Data Register Half

3.5.13 Special Applications

To read consecutive, aligned 16-bit values for high-performance DSP operations, use the Load Data Register instructions instead of these half-word instructions. The half-word load instructions use only half the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.

3.6 Load High Data Register Half

3.6.1 General Form

Dreg_hi = W [indirect_address]

3.6.2 Syntax

Dreg_hi = W [Ireg] ;	// indirect (DAG) (a)
Dreg_hi = W [Ireg ++] ;	// indirect, post-increment (DAG) (a)
Dreg_hi = W [Ireg --] ;	// indirect, post-decrement (DAG) (a)
Dreg_hi = W [Preg] ;	// indirect (a)
Dreg_hi = W [Preg ++ Preg] ;	// indirect, post-increment index [4] (a)

3.6.3 Syntax Terminology

Dreg_hi: the most significant 16 bits of registers R0, ..., R7

Preg: P0, ..., P5, SP, FP

Ireg: I0, ..., I3

3.6.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

3.6.5 Functional Description

The Load High Data Register Half instruction loads 16 bits from a memory location indicated by an I-register or a P-register into the most significant half of a 32-bit data register. The operation does not affect the least significant half.

The indirect address and offset must yield an even numbered address to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

3.6.6 Options

The Load High Data Register Half instruction supports the following options:

- Post-increment the source pointer I-register by 2 bytes to maintain half-word alignment
- Post-decrement the source pointer I-register by 2 bytes to maintain half-word alignment

[4] See "Indirect and Post-Increment Index Addressing" on page 3-16.

3.6.7 Indirect and Post-Increment Index Addressing

The syntax of the form Dst_hi = [Src_1 ++ Src_2] is indirect and post-increment index addressing. The form is shorthand for the following sequence:

```
Dst_hi = [Src_1] ;          /* load the half-word into the upper half of the destination
                            register, indirect */
Src_1 += Src_2 ;            // post-increment Src_1 by a quantity indexed by Src_2
```

where:

- Dst_hi is the most significant half of the destination register. (Dreg_hi in the syntax example).
- Src_1 is the memory source pointer register on the right-hand side of the syntax.
- Src_2 is the increment pointer register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common Preg 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].

3.6.8 Flags Affected

None

3.6.9 Required Mode

User & Supervisor

3.6.10 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

3.6.11 Example

```
r3.h = w [ i1 ] ;
r7.h = w [ i3 ++ ] ;
r1.h = w [ i0 -- ] ;
r2.h = w [ p4 ] ;
r5.h = w [ p2 ++ p0 ] ;
```

3.6.12 Also See

Load Low Data Register Half, Load Half-Word – Zero-Extended, Load Half-Word – Sign-Extended

3.6.13 Special Applications

To read consecutive, aligned 16-bit values for high-performance DSP operations, use the Load Data Register instructions instead of these half-word instructions. The half-word load instructions use only half the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.

3.7 Load Low Data Register Half

3.7.1 General Form

Dreg_lo = W [indirect_address]

3.7.2 Syntax

```
Dreg_lo = W [ Ireg ] ;                  // indirect (DAG) (a)
Dreg_lo = W [ Ireg ++ ] ;               // indirect, post-increment (DAG) (a)
Dreg_lo = W [ Ireg -- ] ;               // indirect, post-decrement (DAG) (a)
Dreg_lo = W [ Preg ] ;                  // indirect (a)
Dreg_lo = W [ Preg ++ Preg ] ;          // indirect, post-increment index [5] (a)
```

3.7.3 Syntax Terminology

Dreg_lo: the least significant 16 bits of registers R0, ..., R7

Preg: P0, ..., P5, SP, FP

Ireg: I0, ..., I3

3.7.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

3.7.5 Functional Description

The Load Low Data Register Half instruction loads 16 bits from a memory location indicated by an I-register or a P-register into the least significant half of a 32-bit data register. The operation does not affect the most significant half of the data register.

The indirect address and offset must yield an even numbered address to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

3.7.6 Options

The Load Low Data Register Half instruction supports the following options:

- Post-increment the source pointer I-register by 2 bytes
- Post-decrement the source pointer I-register by 2 bytes

[5] See “Indirect and Post-Increment Index Addressing” on page 3-19.

3.7.7 Indirect and Post-Increment Index Addressing

The syntax of the form Dst_lo = [Src_1 ++ Src_2] is indirect and post-increment index addressing. The form is shorthand for the following sequence:

Dst_lo = [Src_1] ;	/* load the half-word into the lower half of the destination register, indirect */
Src_1 += Src_2 ;	// post-increment Src_1 by a quantity indexed by Src_2

where:

- Dst_lo is the least significant half of the destination register (Dreg_lo in the syntax example).
- Src_1 is the memory source pointer register on the right side of the syntax.
- Src_2 is the increment index register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common Preg 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].

3.7.8 Flags Affected

None

3.7.9 Required Mode

User & Supervisor

3.7.10 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

3.7.11 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

3.7.12 Example

```
r3.l = w[ i1 ] ;
r7.l = w[ i3 ++ ] ;
r1.l = w[ i0 -- ] ;
r2.l = w[ p4 ] ;
r5.l = w[ p2 ++ p0 ] ;
```

3.7.13 Also See

Load High Data Register Half, Load Half-Word – Zero-Extended, Load Half-Word – Sign-Extended

3.7.14 Special Applications

To read consecutive, aligned 16-bit values for high-performance DSP operations, use the Load Data Register instructions instead of these half-word instructions. The half-word load instructions use only half of the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.

3.8 Load Byte – Zero-Extended

3.8.1 General Form

D-register = B [indirect_address] (Z)

3.8.2 Syntax

```
Dreg = B [ Preg ] (Z);                   // indirect (a)
Dreg = B [ Preg ++ ] (Z);                // indirect, post-increment (a)
Dreg = B [ Preg -- ] (Z);                // indirect, post-decrement (a)
Dreg = B [ Preg + uimm15 ] (Z);          // indexed with offset (b)
Dreg = B [ Preg - uimm15 ] (Z);          // indexed with offset (b)
```

3.8.3 Syntax Terminology

Dreg: R0, ..., R7

Preg: P0, ..., P5, SP, FP

uimm15: 15-bit unsigned field, with a range of 0 through 32,767 bytes (0x0000 through 0x7FFF)

3.8.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

3.8.5 Functional Description

The Load Byte – Zero-Extended instruction loads an 8-bit byte, zero-extended to 32 bits indicated by an I-register or a P-register, from a memory location into a 32-bit data register. Fill the D-register bits 31:8 with zeros.

The indirect address and offset have no restrictions for memory address alignment.

3.8.6 Options

The Load Byte – Zero-Extended instruction supports the following options:

- Post-increment the source pointer by 1 byte
- Post-decrement the source pointer by 1 byte
- Offset the source pointer with a 16-bit signed constant.

3.8.7 Flags Affected

None

3.8.8 Required Mode

User & Supervisor

3.8.9 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

3.8.10 Example

```
r3 = b [  p0 ] (z);
r7 = b [ p1 ++ ] (z);
r2 = b [ sp -- ] (z);
r0 = b [ p4 + 0xFFFF800F ] (z);
```

3.8.11 Also See

Load Byte – Sign-Extended

3.8.12 Special Applications

3.9 Load Byte – Sign-Extended

3.9.1 General Form

D-register = B [indirect_address]

3.9.2 Syntax

```
Dreg = B [ Preg ] (X);                  // indirect (a)
Dreg = B [ Preg ++ ] (X);               // indirect, post-increment (a)
Dreg = B [ Preg -- ] (X) ;              // indirect, post-decrement (a)
Dreg = B [ Preg + uimm15 ] (X) ;        // indexed with offset (b)
Dreg = B [ Preg - uimm15 ] (X) ;        // indexed with offset (b)
```

3.9.3 Syntax Terminology

Dreg: R0, ..., R7

Preg: P0, ..., P5, SP, FP

uimm15: 15-bit unsigned field, with a range of 0 through 32,767 bytes (0x0000 through 0x7FFF)

3.9.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

3.9.5 Functional Description

The Load Byte – Sign-Extended instruction loads an 8-bit byte, sign-extended to 32 bits, from a memory location indicated by a P-register into a 32-bit data register. The pointer register is a P-register. Fill the D-register bits 31:8 with the most significant bit of the loaded byte.

The indirect address and offset have no restrictions for memory address alignment.

3.9.6 Options

Optionally, (X) can be included in the instruction syntax to make it clearer that the load is sign extended. This optional flag does not affect the instruction encoding, function, or performance.

The Load Byte – Sign-Extended instruction supports the following options:

- Post-increment the source pointer by 1 byte
- Post-decrement the source pointer by 1 byte
- Offset the source pointer with a 16-bit signed constant.

3.9.7 Flags Affected

None

3.9.8 Required Mode

User & Supervisor

3.9.9 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

3.9.10 Example

```
r3 = b [  p0 ] (x);
r7 = b [ p1 ++ ](x) ;
r2 = b [ sp -- ] (x);
r0 = b [ p4 + 0xFFFF800F ](x) ;
```

3.9.11 Also See

Load Byte – Zero-Extended

3.9.12 Special Applications

3.10 Store Pointer Register

3.10.1 General Form

[indirect_address] = P-register

3.10.2 Syntax

```
[ Preg ] = Preg ;                      // indirect (a)
[ Preg ++ ] = Preg ;                   // indirect, post-increment (a)
[ Preg -- ] = Preg ;                   // indirect, post-decrement (a)
[ Preg + uimm6m4 ] = Preg ;            // indexed with small offset (a)
[ Preg + uimm17m4 ] = Preg ;           // indexed with large offset (b)
[ Preg - uimm17m4 ] = Preg ;           // indexed with large offset (b)
[ FP - uimm7m4 ] = Preg ;              // indexed FP-relative (a)
```

3.10.3 Syntax Terminology

Preg: P0, ..., P5, SP, FP

uimm6m4: 6-bit unsigned field that must be a multiple of 4, with a range of 0 through 60 bytes

uimm7m4: 7-bit unsigned field that must be a multiple of 4, with a range of 4 through 128 bytes

uimm17m4: 17-bit unsigned field that must be a multiple of 4, with a range of 0 through 131,068 bytes (0x000 0000 through 0x0001 FFFC)

3.10.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

3.10.5 Functional Description

The Store Pointer Register instruction stores the contents of a 32-bit P-register to a 32-bit memory location. The pointer register is a P-register.

The indirect address and offset must yield an even multiple of 4 to maintain 4-byte word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

3.10.6 Options

The Store Pointer Register instruction supports the following options:

- Post-increment the destination pointer by 4 bytes
- Post-decrement the destination pointer by 4 bytes

- Offset the source pointer with a small (6-bit), word-aligned (multiple of 4), unsigned constant.
- Offset the source pointer with a large (18-bit), word-aligned (multiple of 4), signed constant.
- Frame Pointer (FP) relative and offset with a 7-bit, word-aligned (multiple of 4), negative constant.

The indexed FP-relative form is typically used to access local variables in a subroutine or function. Positive offsets relative to FP (such as is useful to access arguments from a called function) can be accomplished using one of the other versions of this instruction. Preg includes the Frame Pointer and Stack Pointer.

3.10.7 Flags Affected

None

3.10.8 Required Mode

User & Supervisor

3.10.9 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

3.10.10 Example

```
[ p2 ]  = p3 ;
[ sp ++ ]  = p5 ;
[ p0 -- ]  = p2 ;
[ p2 + 8 ]  = p3 ;
[ p2 + 0x4444 ]  = p0 ;
[ fp -12 ]  = p1 ;
```

3.10.11 Also See

Push, Push Multiple

3.10.12 Special Applications

3.11 Store Data Register

3.11.1 General Form

[indirect_address] = D-register

3.11.2 Syntax

Using Pointer Registers

```
[ Preg ] = Dreg ;                    // indirect (a)
[ Preg ++ ] = Dreg ;                 // indirect, post-increment (a)
[ Preg -- ] = Dreg ;                 // indirect, post-decrement (a)
[ Preg + uimm6m4 ] = Dreg ;          // indexed with small offset (a)
[ Preg + uimm17m4 ] = Dreg ;         // indexed with large offset (b)
[ Preg - uimm17m4 ] = Dreg ;         // indexed with large offset (b)
[ Preg ++ Preg ] = Dreg ;            // indirect, post-increment index [6] (a)
[ FP - uimm7m4 ] = Dreg ;            // indexed FP-relative (a)
```

Using Data Address Generator (DAG) Registers

```
[ Ireg ]  = Dreg ;                   // indirect (a)
[ Ireg ++ ]  = Dreg ;                // indirect, post-increment (a)
[ Ireg -- ]  = Dreg ;                // indirect, post-decrement (a)
[ Ireg ++ Mreg ]  = Dreg ;           // indirect, post-increment index[6] (a)
```

3.11.3 Syntax Terminology

Dreg: R0, ..., R7

Preg: P0, ..., P5, SP, FP

Ireg: I0, ..., I3

Mreg: M0, ..., M3

uimm6m4: 6-bit unsigned field that must be a multiple of 4, with a range of 0 through 60 bytes

uimm7m4: 7-bit unsigned field that must be a multiple of 4, with a range of 4 through 128 bytes

uimm17m4: 17-bit unsigned field that must be a multiple of 4, with a range of 0 through 131,068 bytes (0x0000 through 0xFFFC)

3.11.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

[6] See "Indirect and Post-Increment Index Addressing" on page 3-28.

3.11.5 Functional Description

The Store Data Register instruction stores the contents of a 32-bit D-register to a 32-bit memory location. The destination pointer register can be a P-register, I-register, or the Frame Pointer.

The indirect address and offset must yield an even multiple of 4 to maintain 4-byte word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

3.11.6 Options

The Store Data Register instruction supports the following options:

- Post-increment the destination pointer by 4 bytes
- Post-decrement the destination pointer by 4 bytes
- Offset the source pointer with a small (6-bit), word-aligned (multiple of 4), unsigned constant.
- Offset the source pointer with a large (18-bit), word-aligned (multiple of 4), signed constant.
- Frame Pointer (FP) relative and offset with a 7-bit, word-aligned (multiple of 4), negative constant.

The indexed FP-relative form is typically used to access local variables in a subroutine or function. Positive offsets relative to FP (such as is useful to access arguments from a called function) can be accomplished using one of the other versions of this instruction. Preg includes the Frame Pointer and Stack Pointer.

3.11.7 Indirect and Post-Increment Index Addressing

The syntax of the form [Dst_1 ++ Dst_2] = Src is indirect and post-increment index addressing. The form is shorthand for the following sequence:

```
[ Dst_1 ] = Src ;           // load the 32-bit source, indirect
Dst_1 += Dst_2 ;            // post-increment Dst_1 by a quantity indexed by Dst_2
```

where:

- Src is the source register (Dreg in the syntax example).
- Dst_1 is the memory destination register on the left side of the equation.
- Dst_2 is the increment index register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common Preg is used for the inputs, the auto-increment feature does not work.

3.11.8 Flags Affected

None

3.11.9 Required Mode

User & Supervisor

3.11.10 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

3.11.11 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 ;
```

3.11.12 Also See

Load Immediate

3.11.13 Special Applications

3.12 Store High Data Register Half

3.12.1 General Form

W [indirect_address] = Dreg_hi

3.12.2 Syntax

```
W [ Ireg ] = Dreg_hi ;                 // indirect (DAG) (a)
W [ Ireg ++ ]  = Dreg_hi ;             // indirect, post-increment (DAG) (a)
W [ Ireg -- ]  = Dreg_hi ;             // indirect, post-decrement (DAG) (a)
W [ Preg ]  = Dreg_hi ;                // indirect (a)
W [ Preg ++ Preg ] = Dreg_hi ;         // indirect, post-increment index [7] (a)
```

3.12.3 Syntax Terminology

Dreg_hi: the most significant 16 bits of registers R0, ..., R7

Preg: P0, ..., P5, SP, FP

Ireg: I0, ..., I3

3.12.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

3.12.5 Functional Description

The Store High Data Register Half instruction stores the most significant 16 bits of a 32-bit data register to a 16-bit memory location. The pointer register is either an I-register or a P-register.

The indirect address and offset must yield an even number to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

3.12.6 Options

The Store High Data Register Half instruction supports the following options:

- Post-increment the destination pointer I-register by 2 bytes
- Post-decrement the destination pointer I-register by 2 bytes

[7] See "Indirect and Post-Increment Index Addressing" on page 3-31.

3.12.7 Indirect and Post-Increment Index Addressing

The syntax of the form [Dst_1 ++ Dst_2] = Src_hi is indirect and post-increment index addressing. The form is shorthand for the following sequence:

```
[Dst_1] = Src_hi ;        // Store the upper half of the source register, indirect
Dst_1 += Dst_2 ;          // Post-increment Dst_1 by a quantity indexed by Dst_2
```

where:

- Src_hi is the most significant half of the source register (Dreg_hi in the syntax example).
- Dst_1 is the memory destination pointer register on the left side of the syntax.
- Dst_2 is the increment index register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common Preg is used for the inputs, the auto-increment feature does not work.

3.12.8 Flags Affected

None

3.12.9 Required Mode

User & Supervisor

3.12.10 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

3.12.11 Example

```
w[ i1 ]  = r3.h ;
w[ i3 ++ ]  = r7.h ;
w[ i0 -- ]  = r1.h ;
w[ p4 ]  = r2.h ;
w[ p2 ++ p0 ]  = r5.h ;
```

3.12.12 Also See

Store Low Data Register Half

3.12.13 Special Applications

To write consecutive, aligned 16-bit values for high-performance DSP operations, use the Store Data Register instructions instead of these half-word instructions. The half-word store instructions use only half the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.

3.13 Store Low Data Register Half

3.13.1 General Form

W [indirect_address] = Dreg_lo

W [indirect_address] = D-register

3.13.2 Syntax

```
W [ Ireg ] = Dreg_lo ;                // indirect (DAG) (a)
W [ Ireg ++ ]  = Dreg_lo ;            // indirect, post-increment (DAG) (a)
W [ Ireg -- ]  = Dreg_lo ;            // indirect, post-decrement (DAG) (a)
W [ Preg ]  = Dreg_lo ;               // indirect (a)
W [ Preg ]  = Dreg ;                  // indirect (a)
W [ Preg ++ ]  = Dreg ;               // indirect, post-increment (a)
W [ Preg -- ] = Dreg ;                // indirect, post-decrement (a)
W [ Preg + uimm5m2 ]  = Dreg ;        // indexed with small offset (a)
W [ Preg + uimm16m2 ] = Dreg ;        // indexed with large offset (b)
W [ Preg - uimm16m2 ] = Dreg ;        // indexed with large offset (b)
W [ Preg ++ Preg ] = Dreg_lo ;        // indirect, post-increment index [8] (a)
```

3.13.3 Syntax Terminology

Dreg_lo: the least significant 16 bits of registers R0, ..., R7

Preg: P0, ..., P5, SP, FP

Ireg: I0, ..., I3

Dreg: R0, ..., R7

uimm5m2: 5-bit unsigned field that must be a multiple of 2, with a range of 0 through 30 bytes

uimm16m2: 16-bit unsigned field that must be a multiple of 2, with a range of 0 through 65,534 bytes (0x0000 through 0xFFFE)

3.13.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

3.13.5 Functional Description

The Store Low Data Register Half instruction stores the least significant 16 bits of a 32-bit data register to a 16-bit memory location. The pointer register is either an I-register or a P-register.

[8] See "Indirect and Post-Increment Index Addressing" on page 3-34.

The indirect address and offset must yield an even number to maintain 2-byte half-word address alignment. Failure to maintain proper alignment causes an misaligned memory access exception.

3.13.6 Options

The Store Low Data Register Half instruction supports the following options:

- Post-increment the destination pointer by 2 bytes
- Post-decrement the destination pointer by 2 bytes
- Offset the source pointer with a small (5-bit), half-word-aligned (even), unsigned constant.
- Offset the source pointer with a large (17-bit), half-word-aligned (even), signed constant.

3.13.7 Indirect and Post-Increment Index Addressing

The syntax of the form [Dst_1 ++ Dst_2] = Src is indirect and post-increment index addressing. The form is shorthand for the following sequence:

```
[Dst_1] = Src_lo;          // store the lower half of the source register, indirect
Dst_1 += Dst_2 ;           // post-increment Dst_1 by a quantity indexed by Dst_2
```

where:

- Src is the least significant half of the source register (Dreg or Dreg_lo in the syntax example).
- Dst_1 is the memory destination pointer register on the left side of the syntax.
- Dst_2 is the increment index register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate P-registers for the input operands. If a common Preg is used for the inputs, the auto-increment feature does not work.

3.13.8 Flags Affected

None

3.13.9 Required Mode

User & Supervisor

3.13.10 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

3.13.11 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 ;
```

3.13.12 Also See

Store High Data Register Half, Store Data Register

3.13.13 Special Applications

To write consecutive, aligned 16-bit values for high-performance DSP operations, use the Store Data Register instructions instead of these half-word instructions. The half-word store instructions use only half the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.

3.14 Store Byte

3.14.1 General Form

B [indirect_address] = D-register

3.14.2 Syntax

```
B [ Preg ]  = Dreg ;               // indirect (a)
B [ Preg ++ ]  = Dreg ;            // indirect, post-increment (a)
B [ Preg -- ] = Dreg ;             // indirect, post-decrement (a)
B [ Preg + uimm15 ] = Dreg ;       // indexed with offset (b)
B [ Preg - uimm15 ] = Dreg ;       // indexed with offset (b)
```

3.14.3 Syntax Terminology

Dreg: R0, ..., R7

Preg: P0, ..., P5, SP, FP

uimm15: 15-bit unsigned field, with a range of 0 through 32,767 bytes (0x0000 through 0x7FFF)

3.14.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

3.14.5 Functional Description

The Store Byte instruction stores the least significant 8-bit byte of a data register to an 8-bit memory location. The pointer register is a P-register.

The indirect address and offset have no restrictions for memory address alignment.

3.14.6 Options

The Store Byte instruction supports the following options:

- Post-increment the destination pointer by 1 byte to maintain byte alignment
- Post-decrement the destination pointer by 1 byte to maintain byte alignment
- Offset the destination pointer with a 16-bit signed constant.

3.14.7 Flags Affected

None

3.14.8 Required Mode

User & Supervisor

3.14.9 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 32-bit versions of this instruction cannot be issued in parallel with other instructions.

3.14.10 Example

```
b [  p0 ]  = r3 ;
b [ p1 ++ ]  = r7 ;
b [ sp -- ]  = r2 ;
b [ p4 + 0x100F ]  = r0 ;
b [ p4 - 0x53F ]  = r0 ;
```

3.14.11 Also See

3.14.12 Special Applications

To write consecutive, 8-bit values for high-performance DSP operations, use the Store Data Register instructions instead of these byte instructions. The byte store instructions use only one eighth the available 32-bit data bus bandwidth, possibly imposing a bottleneck constriction in the data flow rate.

MOVE 4

Instruction Summary

This chapter discusses the move instructions. Users can take advantage of these instructions to move registers (or register-halves), half words (zero- or sign-extended), move bytes, and perform conditional moves.

4.1 Move Register

4.1.1 General Form

dest_reg = src_reg

4.1.2 Syntax

```
allreg = allreg ;                               // (a)
```

Warning: Not all register combinations are allowed. See the Functional Description for details.

```
A0 = A1 ;                                       // 40-bit Accumulator (b)
A1 = A0 ;                                       // 40-bit Accumulator (b)
A0 = Dreg ;                                     // 32-bit D-register to 32-bit A0.W acc. (b)
A1 = Dreg ;                                     // 32-bit D-register to 32-bit A1.W acc. (b)
sysreg = Preg ;                                 // 32-bit P-register to sysreg (a)
```

ACCUMULATOR TO D-REGISTER MOVE

```
Dreg_even = A0 ;                                // move A0 to even Dreg (b)
Dreg_odd = A1 ;                                 // move A1 to odd Dreg (b)
Dreg_even = A0, Dreg_odd = A1 (opt_mode) ;      // move both Accumulators to a register pair (b)
Dreg_odd = A1, Dreg_even = A0 (opt_mode) ;      // move both Accumulators to a register pair (b)
Dreg_even = A0.X ;                               /* 8-bit A0.X, sign-extended, into 32 bits
                                                of Dreg_even (b) */
Dreg_odd = A1.X ;                                /* 8-bit A1.X, sign-extended, into 32 bits
                                                of Dreg_odd (b) */
```

4.1.3 Syntax Terminology

allreg: R0, ..., R7, P0, ..., P5, SP, FP, I0, ..., I3, M0, ..., M3, B0, ..., B3, L0, ..., L3, A0.X, A0.W, A1.X, A1.W, ASTAT, RETS, RETI, RETX, RETN, RETE, LC0 and LC1, LT0 and LT1, LB0 and LB1, EMUDAT, USP, SEQSTAT and SYSCFG

Dreg: R0, ..., R7

Preg: P0, ..., P5, SP, FP

sysreg: ASTAT, SEQSTAT, SYSCFG, RETI, RETX, RETN, RETE, RETS, LC0 and LC1, LT0 and LT1, LB0 and LB1, and EMUDAT

Dreg_even: R0, R2, R4, R6

Dreg_odd: R1, R3, R5, R7

Note: When combining two moves in the same instruction, the Dreg_even and Dreg_odd operands must be members of the same register pair, i.e. from the set R1:0, R3:2, R5:4, R7:6.

opt_mode: Optionally (ISS2)

4.1.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

4.1.5 Functional Description

The Move Register instruction copies the contents of the source register into the destination register. The operation does not affect the source register contents.

Not all register combinations are allowed. Register moves between the following register groups are disallowed:

1. sysreg = sysreg,
2. sysreg = Ireg, Mreg, Breg, or Lreg,
3. Ireg, Mreg, Breg, or Lreg = sysreg,
4. Preg or USP = sysreg,

where “sysreg” includes ASTAT, SEQSTAT, SYSCFG, RETI, RETX, RETN, RETE, RETS, LC0 and LC1, LT0 and LT1, LB0 and LB1, and EMUDAT.

Moves from Preg or USP to sysreg are allowed.

The Accumulator Extension registers A0.X and A1.X are only defined for the 8 low-order bits A0.X[7:0] and A1.X[7:0]. Any move to/from the upper bits A0.X[31:8] or A1.X[31:8] is undefined.

All moves from smaller to larger registers are sign extended.

4.1.6 Options

The Accumulator to Data Register Move instruction supports these options:

Table 4-1. Accumulator to Data Register Move

Option	Accumulator Copy Formatting
Default	32-bit extraction from Accumulator with 32-bit saturation.
(ISS2)	32-bit extraction with scaling and 32-bit saturation. Scales the Accumulator contents (multiplies x2 by a one-place shift left).

If you want to keep the unaltered contents of the Accumulator, use a simple Move instruction to copy A.x or A.w to or from a register.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

4.1.7 Flags Affected

The ASTAT register that contains the flags can be explicitly modified by this instruction.

The Accumulator to D-register Move versions of this instruction affects the following flags.

- V is set if the result written to the D-register file saturates 32 bits; cleared if no saturation. In the case of two simultaneous operations, V represents the logical “OR” of the two.
- VS is set if V is set; unaffected otherwise.
- AZ is set if result is zero; cleared if non-zero. In the case of two simultaneous operations, AZ represents the logical “OR” of the two.
- AN is set if result is negative; cleared if non-negative. In the case of two simultaneous operations, AN represents the logical “OR” of the two.

All other flags are unaffected.

4.1.8 Required Mode

User & Supervisor for most cases.

Explicit accesses to USP, SEQSTAT, SYSCFG, RETI, RETX, RETN and RETE require Supervisor mode. If any of these registers are explicitly accessed from User mode, an Illegal Use of Protected Resource exception occurs.

4.1.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

4.1.10 Example

```
r3 = r0 ;
r7 = p2 ;
r2 = a0.w ;
r2.l = a0.x ;
r0 = a0.x ;
a0 = a1 ;
a1 = a0 ;
a0 = r7  ;                              // move R7 to 32-bit A0.W
a1 = r3  ;                              // move R3 to 32-bit A1.W
retn = p0  ;                            // must be in Supervisor mode
r2 = a0 ;                               // 32-bit move with saturation
r7 = a1 ;                               // 32-bit move with saturation
r0 = a0 (iss2) ;                        /* 32-bit move with scaling, truncation and
                                        saturation */
```

4.1.11 Also See

Load Immediate to initialize registers.

Move Half-Register to move values explicitly into the A0.X and A1.X registers.

Zero Overhead Loop Setup to implicitly access registers LC0, LT0, LB0, LC1, LT1 and LB1.

Call, Raise and Return to implicitly access registers RETI, RETN, and RETS.

Force Exception and Force Emulation to implicitly access registers RETX and RETE.

4.1.12 Special Applications

4.2 Move Conditional

4.2.1 General Form

IF CC dest_reg = src_reg

IF ! CC dest_reg = src_reg

4.2.2 Syntax

```
IF CC DPreg = DPreg ;                    // move if CC = 1 (a)
IF ! CC DPreg = DPreg ;                  // move if CC = 0 (a)
```

4.2.3 Syntax Terminology

DPreg: R0, ..., R7, P0, ..., P5, SP, FP

4.2.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

4.2.5 Functional 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 D-register or P-register.

4.2.6 Flags Affected

None

4.2.7 Required Mode

User & Supervisor

4.2.8 Parallel Issue

The Move Conditional instruction cannot be issued in parallel with other instructions.

4.2.9 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 ;
```

4.2.10 Also See

Compare, Move CC, Negate CC, Conditional Jump

4.2.11 Special Applications

4.3 Move Half-Word – Zero-Extended

4.3.1 General Form

dest_reg = src_reg (Z)

4.3.2 Syntax

```
Dreg = Dreg_lo (Z);                          // (a)
```

4.3.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo: R0.L, ..., R7.L

4.3.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

4.3.5 Functional Description

The Move Half-Word – Zero-Extended 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. Zero-extend the upper half of the destination register. The operation supports only D-registers. Zero extending a signal negative 16-bit number corrupts the number sign and magnitude in most cases. (Trivial exception: 0x8000 has the same magnitude whether viewed as signed or unsigned.)

4.3.6 Flags Affected

The following flags are affected by the Move Half-Word – Zero-Extended instruction:

- AZ is set if result is zero; cleared if non-zero.
- AN is cleared.
- AC0 is cleared.
- V is cleared.

All other flags are unaffected.

4.3.7 Required Mode

User & Supervisor

4.3.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

4.3.9 Example

```
                                          // If r0.l = 0xFFFF
r4 = r0.l (z);                            // Equivalent to r4.l = r0.l and r4.h = 0
                                          // . . . then r4 = 0x0000FFFF
```

4.3.10 Also See

Move Half-Word – Sign-Extended, Move Register Half

4.3.11 Special Applications

4.4 Move Half-Word – Sign-Extended

4.4.1 General Form

dest_reg = src_reg

4.4.2 Syntax

```
Dreg = Dreg_lo (X) ;                                // (a)
```

4.4.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo: R0.L, ..., R7.L

4.4.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

4.4.5 Functional Description

The Move Half-Word – Sign-Extended instruction converts a signed half-word (16 bits) to a signed 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-extends the upper half of the destination register. The operation supports only D-registers.

4.4.6 Options

Optionally, (X) can be included in the instruction syntax to make it clearer that the load is sign extended. This optional flag does not affect the instruction encoding, function, or performance.

4.4.7 Flags Affected

The following flags are affected by the Move Half-Word – Sign-Extended instruction:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is cleared.
- V is cleared.

All other flags are unaffected.

4.4.8 Required Mode

User & Supervisor

4.4.9 Parallel Issue

This instruction cannot be issued in parallel with any other instructions.

4.4.10 Example

```
r4 = r0.l(x) ;
r4 = r0.l ;
```

4.4.11 Also See

Move Half-Word – Zero-Extended, Move Register Half

4.4.12 Special Applications

4.5 Move Register Half

4.5.1 General Form

dest_reg_half = src_reg_half

dest_reg_half = accumulator (opt_mode)

4.5.2 Syntax

A0.X = Dreg_lo ;	// [1]least significant 8 bits of Dreg into A0.X (b)
A1.X = Dreg_lo ;	// [1] least significant 8 bits of Dreg into A1.X (b)
Dreg_lo = A0.X ;	/* 8-bit A0.X, sign-extended, into least significant 16 bits of Dreg (b) */
Dreg_lo = A1.X ;	/* 8-bit A1.X, sign-extended, into least significant 16 bits of Dreg (b) */
A0.L = Dreg_lo ;	/* least significant 16 bits of Dreg into least significant 16 bits of A0.W (b) */
A1.L = Dreg_lo ;	/* least significant 16 bits of Dreg into least significant 16 bits of A1.W (b) */
A0.H = Dreg_hi ;	/* most significant 16 bits of Dreg into most significant 16 bits of A0.W (b) */
A1.H = Dreg_hi ;	/* most significant 16 bits of Dreg into most significant 16 bits of A1.W (b) */

ACCUMULATOR TO HALF D-REGISTER MOVE

Dreg_lo = A0 (opt_mode);	/* move A0 to lower half of Dreg (b) */
Dreg_hi = A1 (opt_mode);	/* move A1 to upper half of Dreg (b) */

4.5.3 Syntax Terminology

Dreg_lo: the least significant 16 bits of registers R0, ..., R7

Dreg_hi: the most significant 16 bits of registers R0, ..., R7

A0.L: the least significant 16 bits of Accumulator A0.W

A1.L: the least significant 16 bits of Accumulator A1.W

A0.H: the most significant 16 bits of Accumulator A0.W

A1.H: the most significant 16 bits of Accumulator A1.W

opt_mode: Optionally (IS), (IU), (T), (S2RND), (ISS2), or (IH)

[1] The Accumulator Extension registers A0.X and A1.X are defined only for the 8 low-order bits A0.X[7:0] and A1.X[7:0]. This instruction truncates the upper byte of Dreg_lo before moving the value into the Accumulator Extension register (A0.X or A1.X).

4.5.4 Instruction Length

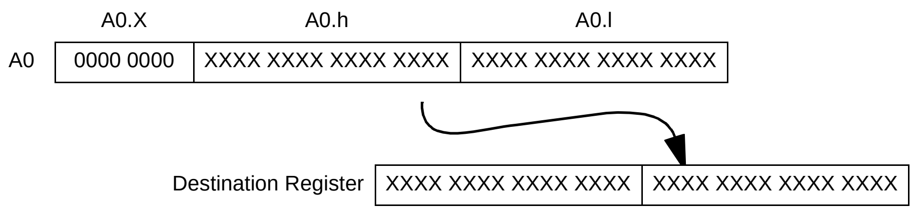
In the syntax, comment (b) identifies 32-bit instruction length.

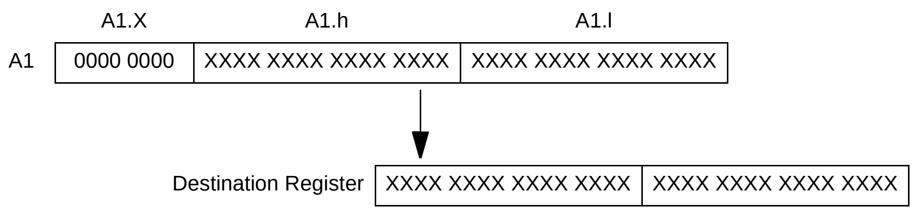
4.5.5 Functional Description

The Move Register Half instruction copies 16 bits from a source register into half of a 32-bit register. The instruction does not affect the unspecified half of the destination register. It supports only D-registers and the Accumulator.

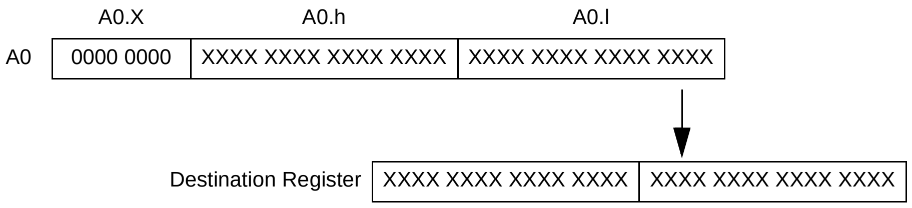
One syntax version simply copies the 16 bits (saturated at 16 bits) of the Accumulator into a data half-register. This syntax supports truncation and rounding beyond a simple Move Register Half instruction.

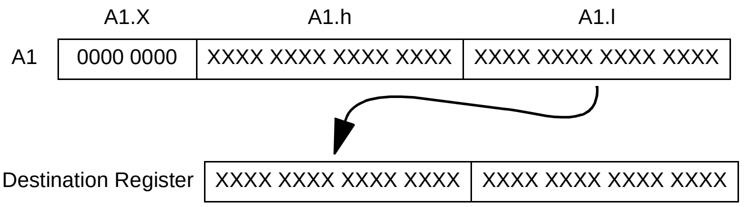
The fraction version of this instruction (the default option) transfers the Accumulator result to the destination register according to the diagrams below.





The integer version of this instruction (the “(IS)” option) transfers the Accumulator result to the destination register according to the diagrams, shown below:





Some versions of this instruction 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. RND_MOD controls rounding for all versions of this instruction except the (IS) and (ISS2) options.

See Section 1.5.6, “Rounding and Truncating,” on page 1-7 for a description of rounding behavior.

4.5.6 Options

The Accumulator to Half D-Register Move instructions support the copy options in Table 4-2.

Table 4-2. Accumulator to Half D-Register Move Options (Sheet 1 of 2)

Option	Accumulator Copy Formatting
Default	High half-word extraction from Accumulator with 16-bit saturation with rounding. Rounding is controlled by RND_MOD bit in the ASTAT register.
(IS)	Low half-word extraction from Accumulator with 16-bit saturation. No rounding.
(IU)	Low half-word extraction from Accumulator with 16-bit saturation with rounding. Rounding is controlled by RND_MOD bit in the ASTAT register.
(T)	High half-word extraction from Accumulator. Truncate low half-word with rounding. Rounding is controlled by RND_MOD bit in the ASTAT register.

Table 4-2. Accumulator to Half D-Register Move Options (Sheet 2 of 2)

Option	Accumulator Copy Formatting
(S2RND)	High half-word extraction with scaling, rounding and 16-bit saturation. Rounding is controlled by RND_MOD bit in the ASTAT register. Scales the Accumulator contents (multiplies x2 by a one-place shift left) and rounds the upper 16 bits before truncating the lower 16 bits.
(ISS2)	Low half-word extraction with scaling and 16-bit saturation. No rounding. Scales the Accumulator contents (multiplies x2 by a one-place shift left) before copying the lower 16-bits.
(IH)	High half-word extraction with 32-bit saturation, then rounding on upper 16-bits. Rounding is controlled by RND_MOD bit in the ASTAT register.

To truncate the result, the operation eliminates the least significant bits that do not fit into the destination register.

When necessary, saturation is performed after the rounding.

If you want to keep the unaltered contents of the Accumulator, use a simple Move instruction to copy A.x or A.w to or from a register.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

4.5.7 Flags Affected

The Accumulator to Half D-register Move versions of this instruction affect the following flags.

- V is set if the result written to the half D-register file saturates 16 bits; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.
- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.

All other flags are unaffected.

4.5.8 Required Mode

User & Supervisor

4.5.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

4.5.10 Example

```
a0.x = r1.l ;
a1.x = r4.l ;
r7.l = a0.x ;
r0.l = a1.x ;
a0.l = r2.l ;
a1.l = r1.l ;
a0.l = r5.l ;
a1.l = r3.l ;
a0.h = r7.h ;
a1.h = r0.h ;
r7.l = a0;                     // copy A0.H into R7.L with saturation.
r2.h = a1;                     // copy A0.H into R2.H with saturation.
r0.h = a1 (is);                // copy A1.L into R0.H with saturation.
r5.l = a0 (t);                 /* copy A0.H into R5.L; truncate A0.L;
                               no saturation.*/
r1.l = a0 (s2rnd);             /* copy A0.H into R1.L with scaling,
                               rounding & saturation.*/
r2.h = a1 (iss2);              /* copy A1.L into R2.H with scaling and
                               saturation.*/
r6.l = a0 (ih);                /* copy A0.H into R6.L with saturation,
                               then rounding. */
```

4.5.11 Also See

Move Half-Word – Zero-Extended, Move Half-Word – Sign-Extended

4.5.12 Special Applications

4.6 Move Byte – Zero-Extended

4.6.1 General Form

dest_reg = src_reg_byte (Z)

4.6.2 Syntax

```
Dreg = Dreg_byte (Z);                    // (a)
```

4.6.3 Syntax Terminology

Dreg_byte: R[7:0].B, the low-order 8 bits of each Data Register

4.6.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

4.6.5 Functional Description

The Move Byte – Zero-Extended instruction converts an unsigned byte to an unsigned word (32 bits). The instruction copies the least significant 8 bits from a source register into the least significant 8 bits of a 32-bit register. The instruction zero-extends the upper bits of the destination register. This instruction supports only D-registers.

4.6.6 Flags Affected

The following flags are affected by the Move Byte – Zero-Extended instruction:

- AZ is set if result is zero; cleared if non-zero.
- AN is cleared.
- AC0 is cleared.
- V is cleared.

All other flags are unaffected.

4.6.7 Required Mode

User & Supervisor

4.6.8 Parallel Issue

This instruction cannot be issued in parallel with any other instructions.

4.6.9 Example

r7 = r2.b (z);

4.6.10 Also See

Move Register Half to explicitly access the Accumulator Extension registers A0.X and A1.X.

Move Byte – Sign-Extended.

4.6.11 Special Applications

4.7 Move Byte – Sign-Extended

4.7.1 General Form

dest_reg = src_reg_byte

4.7.2 Syntax

```
Dreg = Dreg_byte (X) ;                    // (a)
```

4.7.3 Syntax Terminology

Dreg_byte: R[7:0].B, the low-order 8 bits of each Data Register

4.7.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

4.7.5 Functional Description

The Move Byte – Sign-Extended 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 the upper bits of the destination register. This instruction supports only D-registers.

4.7.6 Options

Optionally, (X) can be included in the instruction syntax to make it clearer that the load is sign extended. This optional flag does not affect the instruction encoding, function, or performance.

4.7.7 Flags Affected

The following flags are affected by the Move Byte – Sign-Extended instruction:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is cleared.
- V is cleared.

All other flags are unaffected.

4.7.8 Required Mode

User & Supervisor

4.7.9 Parallel Issue

This instruction cannot be issued in parallel with any other instructions.

4.7.10 Example

```
r7 = r2.b;
r7 = r2.b(x) ;
```

4.7.11 Also See

Move Byte – Zero-Extended

4.7.12 Special Applications

STACK CONTROL 5

Instruction Summary

This chapter discusses the instructions that control the stack. Users can take advantage of these instructions to save the contents of single or multiple registers to the stack or to control the stack frame space on the stack and the Frame Pointer (FP) for that space.

5.1 Push

5.1.1 General Form

[-- SP] = src_reg

5.1.2 Syntax

```
[ -- SP ] = allreg ;                    // predecrement SP (a)
```

5.1.3 Syntax Terminology

allreg: R0, ..., R7, P0, ..., P5, FP, I0, ..., I3, M0, ..., M3, B0, ..., B3, L0, ..., L3, A0.X, A0.W, A1.X, A1.W, ASTAT, RETS, RETI, RETX, RETN, RETE, LC0 and LC1, LT0 and LT1, LB0 and LB1, EMUDAT, USP, SEQSTAT and SYSCFG

5.1.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

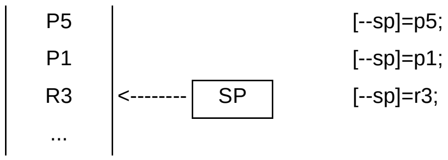
5.1.5 Functional Description

The Push instruction stores the contents of a specified register in the stack. The instruction predecrements the Stack Pointer 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 picture shows what the stack would look like when a series of pushes occur.

higher memory



lower memory

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. (Refer to Section 1.5.1, “Register Names,” on page 1-4 for more information.)

5.1.6 Flags Affected

None

5.1.7 Required Mode

User & Supervisor for most cases.

Explicit accesses to USP, SEQSTAT, SYSCFG, RETI, RETX, RETN and RETE requires Supervisor mode. A protection violation exception results if any of these registers are explicitly accessed from User mode.

5.1.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

5.1.9 Example

```
[ -- sp ] = r0 ;
[ -- sp ] = r1 ;
[ -- sp ] = p0 ;
[ -- sp ] = i0 ;
```

5.1.10 Also See

Push Multiple, Pop

5.1.11 Special Applications

5.2 Push Multiple

5.2.1 General Form

[-- SP] = (src_reg_range)

5.2.2 Syntax

```
[ -- SP ] = ( R7 : Dreglim , P5 : Preglim ) ;      // Dregs and indexed Pregs (a)
[ -- SP ] = ( R7 : Dreglim ) ;                     // Dregs, only (a)
[ -- SP ] = ( P5 : Preglim ) ;                     // indexed Pregs, only (a)
```

5.2.3 Syntax Terminology

Dreglim: any number in the range 7 through 0

Preglim: any number in the range 5 through 0

5.2.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

5.2.5 Functional Description

The Push Multiple instruction saves the contents of multiple data and/or pointer registers to the stack. The range of registers to be saved always includes the highest index register (R7 and/or P5) plus any contiguous lower index registers specified by the user down to and including R0 and/or P0. Push and Push Multiple are the only instructions that perform pre-modify functions.

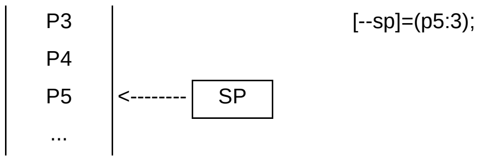
The instructions start by saving the register having the lowest index then advance to the register with the highest index. The index of the first register saved in the stack is specified by the user in the instruction syntax. Data registers are pushed before Pointer registers if both are specified in one instruction.

The instruction predecrements the Stack Pointer to the next available location in the stack first.

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. Therefore, the effective address of the push is SP-4.

The following picture shows what the stack would look like when a push multiple occurs.

higher memory



Lower memory

Because the lowest-indexed registers are saved 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.

Although this instruction takes a variable amount of time to complete depending on the number of registers to be saved, it reduces compiled code size.

This instruction is not interruptible. Interrupts asserted after the first stack write operation is issued are pended until all the writes complete. However, exceptions that occur while this instruction is executing cause it to abort gracefully. For example, a load/store operation might cause a protection violation while Push Multiple is executing. The SP is reset to its value before the execution of this instruction. This measure ensures that the instruction can be restarted after the exception. 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.

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.

Only pointer registers P0, ..., P5 can be operands for this instruction; SP and FP cannot. All data registers R0, ..., R7 can be operands for this instruction.

5.2.6 Flags Affected

None

5.2.7 Required Mode

User & Supervisor

5.2.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

5.2.9 Example

```
[ -- sp ] = (r7:5, p5:0) ;            // D-registers R[4:0] optionally excluded
[ -- sp ] = (r7:2) ;                  // R1:0 excluded
[ -- sp ] = (p5:4) ;                  // P[3:0] excluded
```

5.2.10 Also See

Push, Pop, Pop Multiple

5.2.11 Special Applications

5.3 Pop

5.3.1 General Form

```
dest_reg = [ SP ++ ]
```

5.3.2 Syntax

```
allreg = [ SP ++ ] ;                    // post-increment SP (a)
```

5.3.3 Syntax Terminology

allreg: R0, ..., R7, P0, ..., P5, FP, I0, ..., I3, M0, ..., M3, B0, ..., B3, L0, ..., L3, A0.X, A0.W, A1.X, A1.W, ASTAT, RETS, RETI, RETX, RETN, RETE, LC0 and LC1, LT0 and LT1, LB0 and LB1, EMUDAT, USP, SEQSTAT and SYSCFG

5.3.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

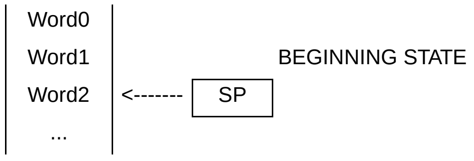
5.3.5 Functional Description

The Pop instruction loads the contents of the stack indexed by the current Stack Pointer into a specified register. The instruction post-increments the Stack Pointer to the next occupied location in the stack before concluding.

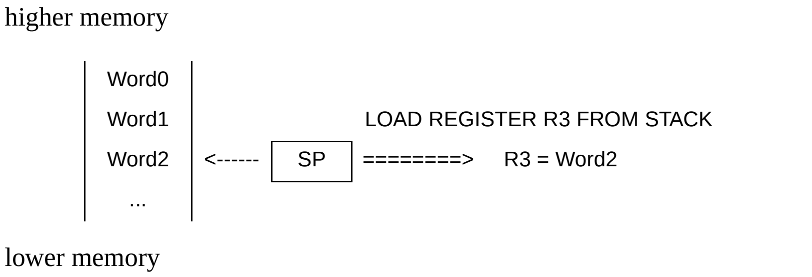
The stack grows down from high memory to low memory, therefore the decrement operation is used for pushing, and the increment operation is used for popping values. The Stack Pointer always points to the last used location. When a pop operation is issued the value pointed to by the Stack Pointer is transferred and the SP is replaced by SP+4.

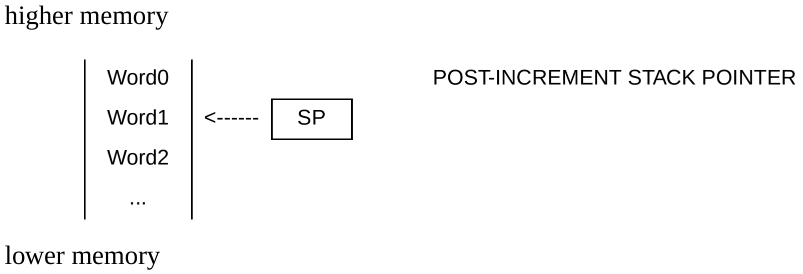
The following picture shows what the stack would look like when a pop such as R3 = [SP ++] occurs.

higher memory



lower memory





The value just popped remains on the stack until another push instruction overwrites it.

Of course, the usual intent for pop 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.

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.

A value cannot be popped off the stack directly into the Stack Pointer. SP = [SP ++] is an invalid instruction. Refer to Section 1.5.1, “Register Names,” on page 1-4 for more information.

5.3.6 Flags Affected

None

5.3.7 Required Mode

User & Supervisor for most cases.

Explicit accesses to USP, SEQSTAT, SYSCFG, RETI, RETX, RETN and RETE requires Supervisor mode. A protection violation exception results if any of these registers are explicitly accessed from User mode.

5.3.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

5.3.9 Example

```
r0 = [sp++] ;
p4 = [sp++] ;
i1 = [sp++] ;
reti = [sp++] ;                                          // supervisor mode required
```

5.3.10 Also See

Push, Push Multiple, Pop Multiple

5.3.11 Special Applications

5.4 Pop Multiple

5.4.1 General Form

(dest_reg_range) = [SP ++]

5.4.2 Syntax

```
( R7 : Dreglim , P5 : Preglim ) = [ SP ++ ] ;     // Dregs and indexed Pregs (a)
( R7 : Dreglim ) = [ SP ++ ] ;                    // Dregs, only (a)
( P5 : Preglim ) = [ SP ++ ] ;                    // indexed Pregs, only (a)
```

5.4.3 Syntax Terminology

Dreglim: any number in the range 7 through 0

Preglim: any number in the range 5 through 0

5.4.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

5.4.5 Functional Description

The Pop Multiple instruction restores the contents of multiple data and/or pointer registers from the stack. The range of registers to be restored always includes the highest index register (R7 and/or P5) plus any contiguous lower index registers specified by the user down to and including R0 and/or P0.

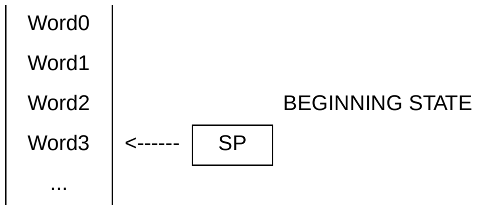
The instructions start by restoring the register having the highest index then descend to the register with the lowest index. The index of the last register restored from the stack is specified by the user in the instruction syntax. Pointer registers are popped before Data registers if both are specified 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.

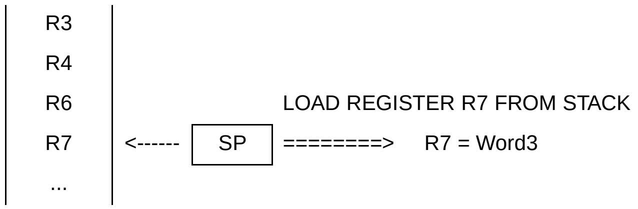
The following pictures show what the stack would look like when a Pop Multiple such as (R7:5) = [SP ++] occurs.

higher memory



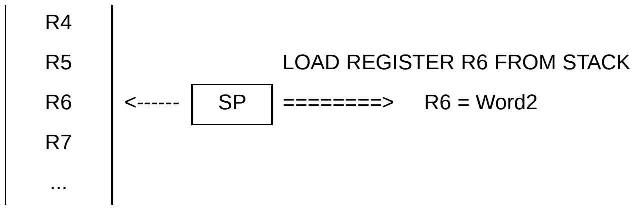
lower memory

higher memory



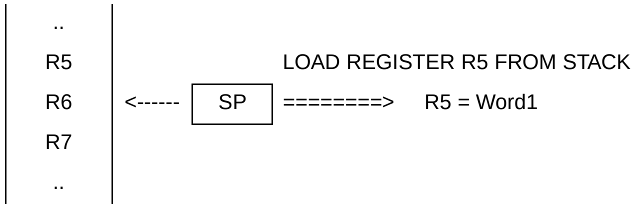
lower memory

higher memory



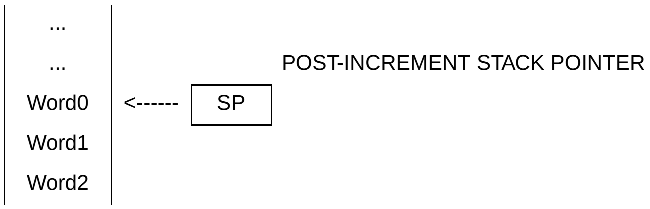
lower memory

higher memory.



lower memory

higher memory



lower memory

The value(s) just popped remain on the stack until another push instruction overwrites it.

Of course, the usual intent for Pop Multiple 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.

Although this instruction takes a variable amount of time to complete depending on the number of registers to be saved, it reduces compiled code size.

This instruction is not interruptible. Interrupts asserted after the first stack read operation is issued are pended until all the reads complete. However, exceptions that occur while this instruction is executing cause it to abort gracefully. For example, a load/store operation might cause a protection violation while Pop Multiple is executing. In that case, SP is reset to its original value prior to the execution of this instruction. This measure ensures that the instruction can be restarted after the exception.

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.

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.

Only pointer registers P0, ..., P5 can be operands for this instruction; SP and FP cannot. All data registers R0, ..., R7 can be operands for this instruction.

5.4.6 Flags Affected

None

5.4.7 Required Mode

User & Supervisor

5.4.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

5.4.9 Example

```
(p5:4) = [ sp ++ ] ;                  // P[3:0] excluded
(r7:2) = [ sp ++ ]  ;                 // R1:0 excluded
(r7:5, p5:0) = [ sp ++ ]  ;           // D-registers R[4:0] optionally excluded
```

5.4.10 Also See

Push, Push Multiple, Pop

5.4.11 Special Applications

5.5 Linkage

5.5.1 General Form

LINK, UNLINK

5.5.2 Syntax

```
LINK uimm18m4 ;                     // allocate a stack frame of specified size (b)
UNLINK ;                            // de-allocate the stack frame (b)
```

5.5.3 Syntax Terminology

uimm18m4: 18-bit unsigned field that must be a multiple of 4, with a range of 8 through 262,152 bytes (0x00008 through 0x3FFFC)

5.5.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

5.5.5 Functional 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 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 $2^{18} + 8 = 262152$ 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 instructions that restores pointer and data registers previously saved to the stack.

The frame values remain on the stack until a subsequent Push, push Push Multiple or LINK operation overwrites them.

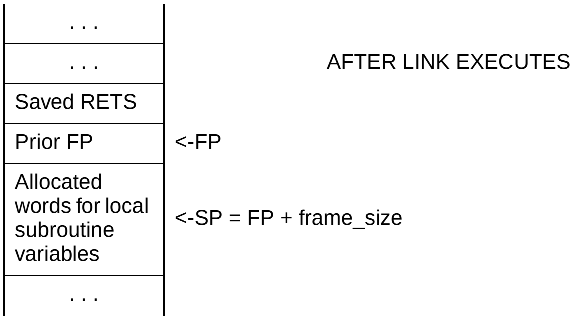
Of course, FP must not be modified by user code between LINK and UNLINK to preserve stack integrity.

Neither LINK nor UNLINK can be interrupted. However, exceptions that occur while either of these instructions is executing causes the instruction to abort. For example, a load/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. Likewise, an aborted UNLINK operation may leave the FP and RETS registers changed because of a load that has already completed before the interruption.

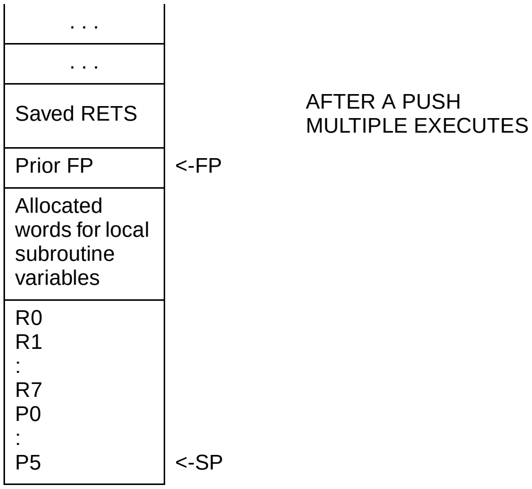
The figures below show the stack contents after executing a LINK instruction followed by a Push Multiple instruction.

higher memory



lower memory

higher memory


. . .
. . .
Saved RETS
AFTER A PUSH MULTIPLE EXECUTES
Prior FP
<-FP
Allocated words for local subroutine variables
R0
R1
:
R7
P0
:
P5
<-SP

lower memory

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.

5.5.6 Flags Affected

None

5.5.7 Required Mode

User & Supervisor

5.5.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

5.5.9 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
```

5.5.10 Also See

Push Multiple, Pop Multiple

5.5.11 Special Applications

The linkage instruction is used to setup and tear down stack frames for a high-level language like C.

CONTROL CODE BIT MANAGEMENT 6

Instruction Summary

This chapter discusses the instructions that affect the Control Code (CC) bit in the ASTAT register. Users can take advantage of these instructions to set the CC bit based on a comparison of values from two registers, pointers, or accumulators. In addition, these instructions can move the status of the CC bit to and from a data register or arithmetic status bit, or they can negate the status of the CC bit.

6.1 Compare Data Register

6.1.1 General Form

CC = operand_1 == operand_2

CC = operand_1 < operand _2

CC = operand _1 <= operand _2

CC = operand _1 < operand _2 (IU)

CC = operand _1 <= operand _2 (IU)

6.1.2 Syntax

```
CC = Dreg == Dreg ;                   // equal, register, signed (a)
CC = Dreg == imm3 ;                   // equal, immediate, signed (a)
CC = Dreg < Dreg ;                    // less than, register, signed (a)
CC = Dreg < imm3 ;                    // less than, immediate, signed (a)
CC = Dreg <= Dreg ;                   // less than or equal, register, signed (a)
CC = Dreg <= imm3 ;                   // less than or equal, immediate, signed (a)
CC = Dreg < Dreg ( IU ) ;             // less than, register, unsigned (a)
CC = Dreg < uimm3 ( IU ) ;            // less than, immediate, unsigned (a)
CC = Dreg <= Dreg ( IU ) ;            // less than or equal, register, unsigned (a)
CC = Dreg <= uimm3 ( IU ) ;           // less than or equal, immediate unsigned (a)
```

6.1.3 Syntax Terminology

Dreg: R0, ..., R7

imm3: 3-bit signed field, with a range of -4 through 3

uimm3: 3-bit unsigned field, with a range of 0 through 7

6.1.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

6.1.5 Functional Description

The Compare Data Register instruction sets the CC Control Code bit based on a comparison of two values. The input operands are D-registers.

The compare operations are non-destructive on the input operands and affect only the CC bit and the flags. 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) syntax is used. 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.

6.1.6 Flags Affected

The Compare Data Register instruction uses the following values in signed and unsigned compare operations:

Comparison	Signed	Unsigned
Equal	AZ=1	n/a
Less than	AN=1	AC=0
Less than or equal	AN or AZ=1	AC=0 or AZ=1

The following flags are affected by the Compare Data Register instruction:

- CC is set if the test condition is true; cleared if false.
- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is set if result generated a carry; cleared if no carry.

All other flags are unaffected.

6.1.7 Required Mode

User & Supervisor

6.1.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

6.1.9 Example

```
cc = r3 == r2 ;
cc = r7 == 1 ;
                                /* If r0 = 0x8FFF FFFF and r3 = 0x0000 0001,
                                 then the unsigned operation . . . */
cc = r0 < r3 ;
                                /* . . . produces cc = 1, because r0 is treated as
                                a negative value */
cc = r2 < -4 ;
cc = r6 <= r1 ;
cc = r4 <= 3  ;
                                /* If r0 = 0x8FFF FFFF and r3 = 0x0000 0001,
                                then the unsigned operation . . .
cc = r5 < r3 (iu) ;
                                /* . . . produces CC = 0, because r0 is treated as
                                a large unsigned value */
cc = r1 < 0x7 (iu) ;
cc = r2 <= r0 (iu) ;
cc = r3 <= 2 (iu) ;
```

6.1.10 Also See

Compare Pointer, Compare Accumulator, Conditional Jump

6.1.11 Special Applications

6.2 Compare Pointer

6.2.1 General Form

CC = operand_1 == operand_2

CC = operand_1 < operand _2

CC = operand _1 <= operand _2

CC = operand _1 < operand _2 (IU)

CC = operand _1 <= operand _2 (IU)

6.2.2 Syntax

```
CC = Preg == Preg ;                // equal, register, signed (a)
CC = Preg == imm3 ;                // equal, immediate, signed (a)
CC = Preg < Preg ;                 // less than, register, signed (a)
CC = Preg < imm3 ;                 // less than, immediate, signed (a)
CC = Preg <= Preg ;                // less than or equal, register, signed (a)
CC = Preg <= imm3 ;                // less than or equal, immediate, signed (a)
CC = Preg < Preg ( IU )  ;         // less than, register, unsigned (a)
CC = Preg < uimm3 ( IU )  ;        // less than, immediate, unsigned (a)
CC = Preg <= Preg ( IU )  ;        // less than or equal, register, unsigned (a)
CC = Preg <= uimm3 ( IU )  ;       // less than or equal, immediate unsigned (a)
```

6.2.3 Syntax Terminology

Preg: P0, ..., P5, SP, FP

imm3: 3-bit signed field, with a range of -4 through 3

uimm3: 3-bit unsigned field, with a range of 0 through 7

6.2.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

6.2.5 Functional Description

The Compare Pointer instruction sets the CC Control Code bit based on a comparison of two values. The input operands are P-registers.

The compare operations are non-destructive on the input operands and affect only the CC bit and the flags. 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) syntax is used. 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.

6.2.6 Flags Affected

- CC is set if the test condition is true; cleared if false.

All other flags are unaffected.

6.2.7 Required Mode

User & Supervisor

6.2.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

6.2.9 Example

```
cc = p3 == p2 ;
cc = p0 == 1 ;
cc = p0 < p3 ;
cc = p2 < -4 ;
cc = p1 <= p0 ;
cc = p4 <= 3  ;
cc = p5 < p3 (iu) ;
cc = p1 < 0x7 (iu) ;
cc = p2 <= p0 (iu) ;
cc = p3 <= 2 (iu) ;
```

6.2.10 Also See

Compare Data Register, Compare Accumulator, Conditional Jump

6.2.11 Special Applications

6.3 Compare Accumulator

6.3.1 General Form

CC = A0 == A1

CC = A0 < A1

CC = A0 <= A1

6.3.2 Syntax

```
CC = A0 == A1 ;                // equal, signed (a)
CC = A0 < A1 ;                 // less than, Accumulator, signed (a)
CC = A0 <= A1 ;                // less than or equal, Accumulator, signed (a)
```

6.3.3 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

6.3.4 Functional Description

The Compare Accumulator instruction sets the CC Control Code 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 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.

No unsigned compare operations nor immediate compare operations are performed for the Accumulators.

The compare operations are non-destructive on the input operands, and affect only the CC bit and the flags. All subsequent conditional branching is based on the value of the CC bit.

6.3.5 Flags Affected

The Compare Accumulator instruction uses the following values in compare operations:

Comparison	Signed
Equal	AZ=1
Less than	AN=1
Less than or equal	AN or AZ=1

The following arithmetic status bits reside in the ASTAT register:

- CC is set if the test condition is true; cleared if false.
- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is set if result generated a carry; cleared if no carry.

All other flags are unaffected.

6.3.6 Required Mode

User & Supervisor

6.3.7 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

6.3.8 Example

```
cc = a0 == a1 ;
cc = a0 < a1 ;
cc = a0 <= a1 ;
```

6.3.9 Also See

Compare Pointer, Compare Data Register, Conditional Jump

6.3.10 Special Applications

6.4 Move CC

6.4.1 General Form

dest = CC

dest |= CC

dest &= CC

dest ^= CC

CC = source

CC |= source

CC &= source

CC ^= source

6.4.2 Syntax

```
Dreg = CC ;               // CC into 32-bit data register, zero-extended (a)
statbit = CC ;            // status bit equals CC (a)
statbit |= CC ;           // status bit equals status bit OR CC (a)
statbit &= CC ;           // status bit equals status bit AND CC (a)
statbit ^= CC  ;          // status bit equals status bit XOR CC (a)
CC = Dreg ;               // CC set if the register is non-zero (a)
CC = statbit ;            // CC equals status bit (a)
CC |= statbit ;           // CC equals CC OR status bit (a)
CC &= statbit ;           // CC equals CC AND status bit (a)
CC ^= statbit ;           // CC equals CC XOR status bit (a)
```

6.4.3 Syntax Terminology

Dreg: R0, ..., R7

statbit: AZ, AN, AC0, AC1, V, VS, AV0, AV0S, AV1, AV1S, AQ

6.4.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

6.4.5 Functional Description

The Move CC instruction moves the status of the Control Code (CC) bit to and from a data register or arithmetic status bit.

When copying the CC bit into a 32-bit register, the operation moves the CC into the least significant bit of the register, zero-extended to 32 bits. The two cases are as follows:

If CC = 0, Dreg becomes 0x00000000.

If CC = 1, Dreg becomes 0x00000001.

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 non-zero. Otherwise, the operation clears the CC bit.

Some versions of this instruction logically set or clear an arithmetic status bit based on the status of the Control Code.

The use of the CC bit as source and destination in the same instruction is disallowed. See the NEGATE CC instruction to change CC based solely on its own value.

6.4.6 Flags Affected

The Move CC instruction affects flags CC, AZ, AN, AC0, AC1, V, VS, AV0, AV0S, AV1, AV1S, AQ, according to the status bit and syntax used, as described in “Syntax” on page 6-9. All other flags not explicitly specified by the syntax are unaffected.

6.4.7 Required Mode

User & Supervisor

6.4.8 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

6.4.9 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

6.4.10 Example

```
r0 = cc ;
az = cc ;
an |= cc ;
ac0 &= cc ;
av0 ^= cc ;
cc = r4 ;
cc = av1 ;
cc |= aq ;
cc &= an ;
cc ^= ac1 ;
```

6.4.11 Also See

NEGATE CC

6.4.12 Special Applications

6.5 Negate CC

6.5.1 General Form

CC = ! CC

6.5.2 Syntax

```
CC = ! CC ;                                   // (a)
```

6.5.3 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

6.5.4 Functional Description

The Negate CC instruction inverts the logical state of CC.

6.5.5 Flags Affected

CC is toggled from its previous value by the Negate CC instruction. All other flags are unaffected.

6.5.6 Required Mode

User & Supervisor

6.5.7 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

6.5.8 Example

```
cc=!cc ;
```

6.5.9 Also See

Move CC

6.5.10 Special Applications

LOGICAL OPERATIONS 7

Instruction Summary

This chapter discusses the instructions that specify logical operations. Users can take advantage of these instructions to perform logical AND, NOT, OR, exclusive-OR, and bit-wise, exclusive-OR operations.

7.1 AND

7.1.1 General Form

dest_reg = src_reg_0 & src_reg_1

7.1.2 Syntax

```
Dreg = Dreg & Dreg ;                    // (a)
```

7.1.3 Syntax Terminology

Dreg: R0, ..., R7

7.1.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

7.1.5 Functional Description

The AND instruction performs a 32-bit, bit-wise logical AND operation on the two source registers and stores the results into the dest_reg.

The instruction does not implicitly modify the source registers. The dest_reg and one src_reg can be the same D-register. So doing explicitly modifies the src_reg.

7.1.6 Flags Affected

The AND instruction affects flags as follows:

- AZ is set if the final result is zero, cleared if non-zero.
- AN is set if the result is negative, cleared if non-negative.
- AC and AV0 are cleared.

All other flags are unaffected.

7.1.7 Required Mode

User & Supervisor

7.1.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

7.1.9 Example

```
r4 = r4 & r3 ;
```

7.1.10 Also See

OR

7.1.11 Special Applications

7.2 NOT (1’s Complement)

7.2.1 General Form

dest_reg = ~ src_reg

7.2.2 Syntax

```
Dreg = ~ Dreg ;                                   // (a)
```

7.2.3 Syntax Terminology

Dreg: R0, ..., R7

7.2.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

7.2.5 Functional Description

The NOT (1’s Complement) instruction toggles every bit in the 32-bit register.

The instruction does not implicitly modify the src_reg. The dest_reg and src_reg can be the same D-register. Using the same D-register as the dest_reg and src_reg would explicitly modify the src_reg at your discretion.

7.2.6 Flags Affected

The NOT (1’s Complement) instruction affects flags as follows:

- AZ is set if the final result is zero, cleared if non-zero.
- AN is set if the result is negative, cleared if non-negative.
- AC and AV0 are cleared.

All other flags are unaffected.

7.2.7 Required Mode

User & Supervisor

7.2.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

7.2.9 Example

```
r3 = ~ r4 ;
```

7.2.10 Also See

Negate (Two’s Complement)

7.2.11 Special Applications

7.3 OR

7.3.1 General Form

dest_reg = src_reg_0 | src_reg_1

7.3.2 Syntax

```
Dreg = Dreg | Dreg ;                              // (a)
```

7.3.3 Syntax Terminology

Dreg: R0, ..., R7

7.3.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

7.3.5 Functional Description

The OR instruction performs a 32-bit, bit-wise logical OR operation on the two source registers and stores the results into the dest_reg.

The instruction does not implicitly modify the source registers. The dest_reg and one src_reg can be the same D-register. So doing explicitly modifies the src_reg.

7.3.6 Flags Affected

The OR instruction affects flags as follows:

- AZ is set if the final result is zero, cleared if non-zero.
- AN is set if the result is negative, cleared if non-negative.
- AC and AV0 are cleared.

All other flags are unaffected.

7.3.7 Required Mode

User & Supervisor

7.3.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

7.3.9 Example

```
r4 = r4 | r3 ;
```

7.3.10 Also See

Exclusive-OR, Bit-Wise Exclusive-OR

7.3.11 Special Applications

7.4 Exclusive-OR

7.4.1 General Form

dest_reg = src_reg_0 ^ src_reg_1

7.4.2 Syntax

```
Dreg = Dreg ^ Dreg ;                              // (a)
```

7.4.3 Syntax Terminology

Dreg: R0, ..., R7

7.4.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

7.4.5 Functional Description

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 dest_reg.

The XOR instruction does not implicitly modify source registers. The dest_reg and one src_reg can be the same D-register. So doing explicitly modifies the src_reg.

7.4.6 Flags Affected

The XOR instruction affects flags as follows:

- AZ is set if the final result is zero, cleared if non-zero.
- AN is set if the result is negative, cleared if non-negative.
- AC and AV0 are cleared.

All other flags are unaffected.

7.4.7 Required Mode

User & Supervisor

7.4.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

7.4.9 Example

```
r4 = r4 ^ r3 ;
```

7.4.10 Also See

OR, Bit-Wise Exclusive-OR

7.4.11 Special Applications

7.5 Bit-Wise Exclusive-OR

7.5.1 General Form

```
dest_reg = CC = BXORSHIFT ( A0, src_reg )
dest_reg = CC = BXOR ( A0, src_reg )
dest_reg = CC = BXOR ( A0, A1, CC )
A0 = BXORSHIFT ( A0, A1, CC )
```

7.5.2 Syntax

LFSR TYPE I (WITHOUT FEEDBACK)

```
Dreg_lo = CC = BXORSHIFT ( A0, Dreg ) ;     // (b)
Dreg_lo = CC = BXOR ( A0, Dreg ) ;          // (b)
```

LFSR TYPE I (WITH FEEDBACK)

```
Dreg_lo = CC = BXOR ( A0, A1, CC ) ;        // (b)
A0 = BXORSHIFT ( A0, A1, CC ) ;             // (b)
```

7.5.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo: RL[7:0]

7.5.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

7.5.5 Functional Description

Four Bit-wise Exclusive-OR (BXOR) instructions support two different types of 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.

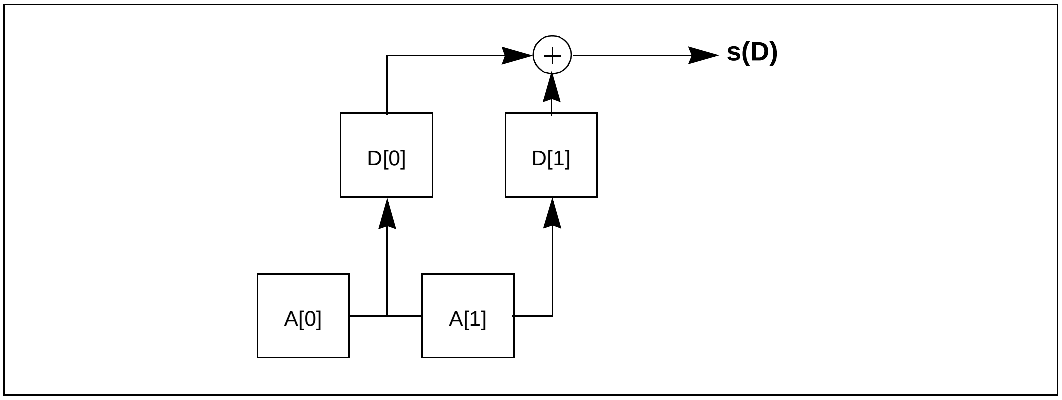
The Type I LFSRs (with feedback) applies a 40-bit mask in Accumulator A1 to a 40-bit state residing in A0. The result is shifted into A0.

In the following circuits describing the BXOR instruction group, a bit-wise XOR reduction is defined as:

$$\text{Out} = ((((B_0 \oplus B_1) \oplus B_2) \oplus B_1)... \oplus B_{n-1})$$

where B_0 through B_{N-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 7-1, “Bit-Wise Exclusive OR Reduction” .

Figure 7-1. Bit-Wise Exclusive OR Reduction



In the figure above, the bits A0[0] and A0[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:

$$s(D) = (A0[0] \bullet D[0]) \oplus (A0[1] \bullet D[1])$$

7.5.5.1 Modified Type I LSFR (without feedback)

Two instructions support the LSFR with no feedback. They are:

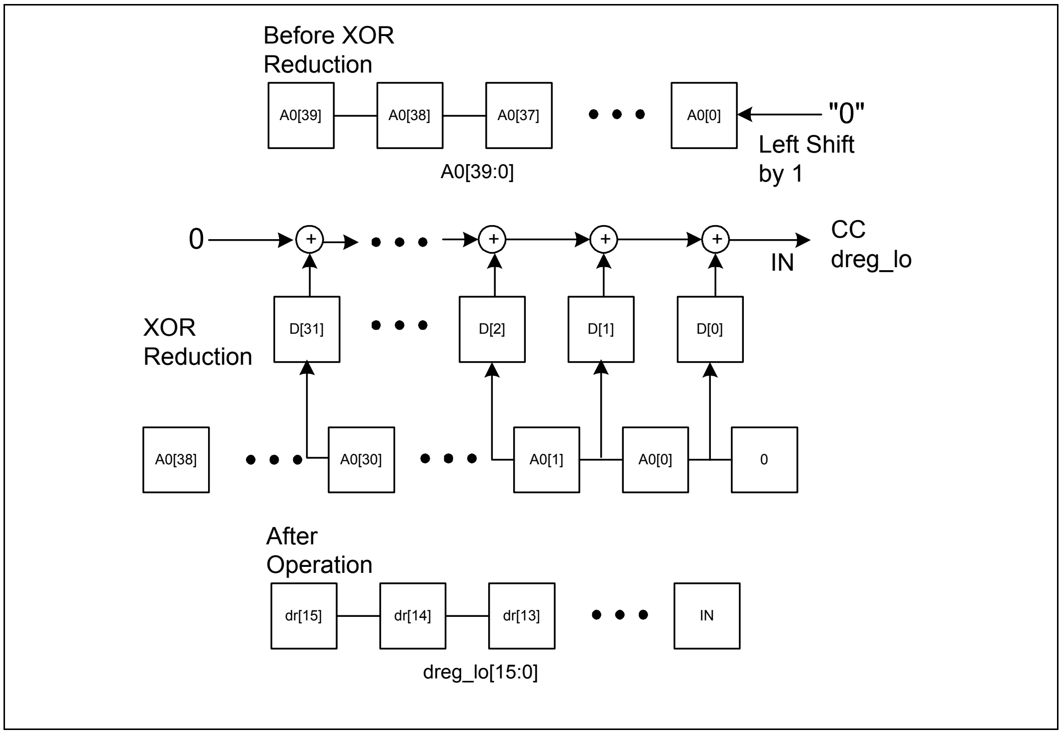
Dreg_lo = BXORSHIFT(A0,dreg), and

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 bitwise XOR of A0 logically AND’ed with a dreg. The result of the operation is placed into both the CC flag and the least significant bit of the destination register. The operation is shown below in Figure 7-2.

The upper 15 bits of dreg_lo are overwritten with zero, and dr[0] = IN after the operation.

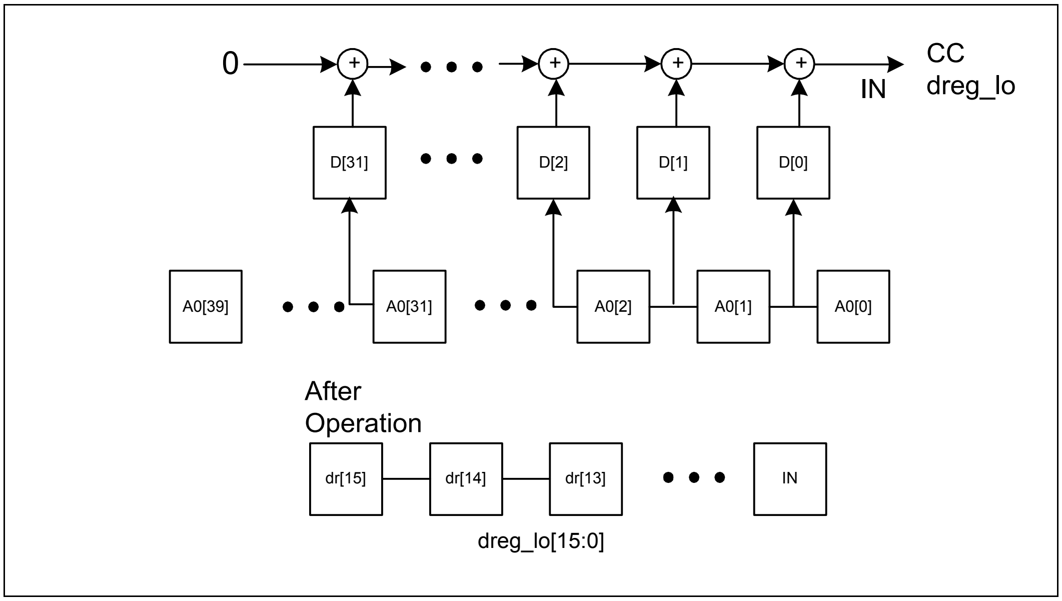
Figure 7-2. A0 Left Shifted by 1 Followed by XOR Reduction



The second instruction in this class performs a bitwise 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 7-3.

The upper 15 bits of dreg_lo are overwritten with zero, and dr[0] = IN after the operation.

Figure 7-3. XOR of A0, Logical AND with the D-Register



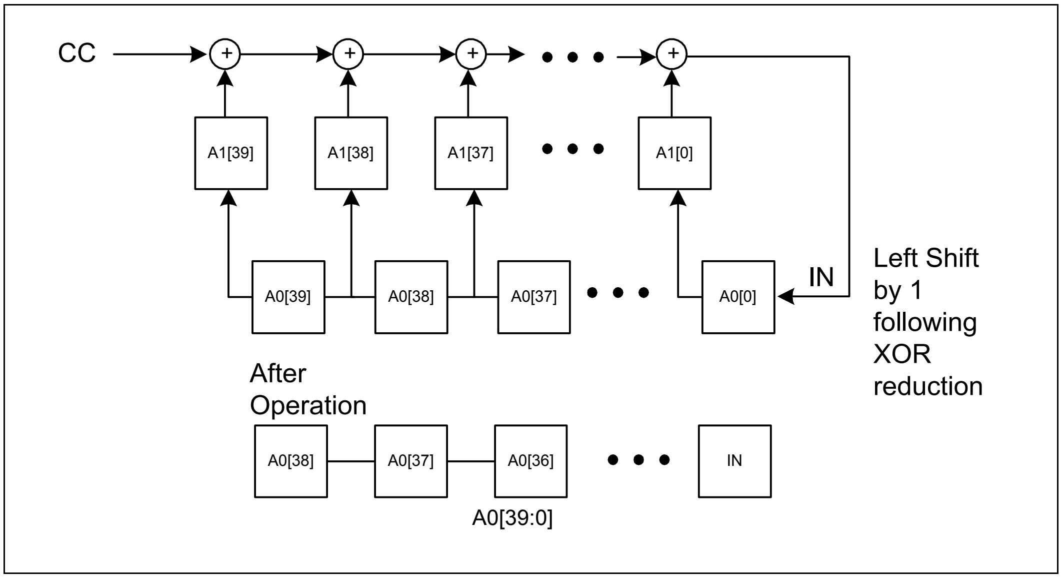
7.5.5.2 Modified Type I LFSR (with feedback)

Two instructions support the LFSR with feedback. They are:

A0 = BXORSHIFT(A0,A1,CC), and
Dreg_lo = CC = BXOR(A0,A1,CC)

The first instruction provides a bitwise XOR of A0 logically AND'ed with A1. The resulting intermediate bit is XOR'ed with the CC flag. The result of the operation is left-shifted into the least significant bit of A0 following the operation. This operation is illustrated in Figure 7-4. The CC bit is not modified by this operation.

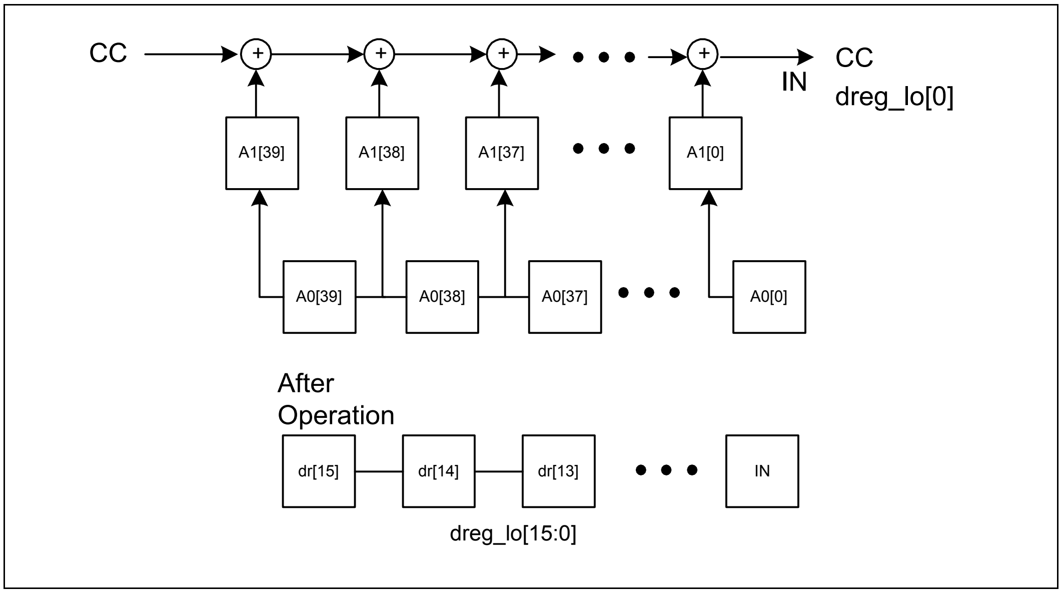
Figure 7-4. XOR of A0 logical AND with A1 with Results Left-Shifted into LSB of A0



The second instruction in this class performs a bitwise XOR of A0 logically AND'ed with A1. The resulting intermediate bit is XOR'ed with the CC flag. The result of the operation is placed into both the CC flag and the least significant bit of the destination register.

This operation is illustrated in Figure 7-5.

Figure 7-5. XOR of A0 Logical AND with A1 with Results Placed in CC Flag and LSB of Destination 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.

7.5.6 Flags Affected

The following flags are affected by the Four Bit-wise Exclusive-OR instruction:

- CC is set or cleared according to the Functional Description for the BXOR and the non-feedback version of the BXORSHIFT instruction. The feedback version of the BXORSHIFT instruction affects no flags.

All other flags are unaffected.

7.5.7 Required Mode

User & Supervisor

7.5.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

7.5.9 Example

```
r0.l = cc = bxorshift (a0, r1) ;
r0.l = cc = bxor (a0, r1) ;
r0.l = cc = bxor (a0, a1, cc) ;
a0 = bxorshift (a0, a1, cc) ;
```

7.5.10 Also See

7.5.11 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.

BIT OPERATIONS 8

Instruction Summary

This chapter discusses the instructions that specify bit operations. Users can take advantage of these instructions to set, clear, toggle, and test bits. They can also merge bit fields and save the result, extract specific bits from a register, merge bit streams, and count the number of 1's in a register.

8.1 Bit Clear

8.1.1 General Form

BITCLR (register, bit_position)

8.1.2 Syntax

```
BITCLR ( Dreg , uimm5 ) ;                        // (a)
```

8.1.3 Syntax Terminology

Dreg: R0, ..., R7

uimm5: 5-bit unsigned field, with a range of 0 through 31

8.1.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

8.1.5 Functional Description

The Bit Clear instruction clears the bit designated by bit_position in the specified D-register. It does not affect other bits in that register.

The bit_position range of values is 0 – 31, where 0 indicates the LSB and 31, the MSB of the 32-bit D-register.

8.1.6 Flags Affected

The Bit Clear instruction affects flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is cleared.
- V is cleared.

All other flags are unaffected.

8.1.7 Required Mode

User & Supervisor

8.1.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

8.1.9 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.

8.1.10 Also See

Bit Set, Bit Test, Bit Toggle

8.1.11 Special Applications

8.2 Bit Set

8.2.1 General Form

BITSET (register, bit_position)

8.2.2 Syntax

```
BITSET ( Dreg , uimm5 ) ;          // (a)
```

8.2.3 Syntax Terminology

Dreg: R0, ..., R7

uimm5: 5-bit unsigned field, with a range of 0 through 31

8.2.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

8.2.5 Functional Description

The Bit Set instruction sets the bit designated by bit_position in the specified D-register. It does not affect other bits in the D-register.

The bit_position range of values is 0 – 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

8.2.6 Flags Affected

The Bit Set instruction sets the AN flag as follows:

- AZ is cleared.
- AN is set if result is negative; cleared if non-negative.
- AC0 is cleared.
- V is cleared.

All other flags are unaffected.

8.2.7 Required Mode

User & Supervisor

8.2.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

8.2.9 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.

8.2.10 Also See

Bit Clear, Bit Test, Bit Toggle

8.2.11 Special Applications

8.3 Bit Toggle

8.3.1 General Form

BITTGL (register, bit_position)

8.3.2 Syntax

```
BITTGL ( Dreg , uimm5 ) ;          // (a)
```

8.3.3 Syntax Terminology

Dreg: R0, ..., R7

uimm5: 5-bit unsigned field, with a range of 0 through 31

8.3.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

8.3.5 Functional Description

The Bit Toggle instruction inverts the bit designated by bit_position in the specified D-register. The instruction does not affect other bits in the D-register.

The bit_position range of values is 0 – 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

8.3.6 Flags Affected

The Bit Toggle instruction affects flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is cleared.
- V is cleared.

All other flags are unaffected.

8.3.7 Required Mode

User & Supervisor

8.3.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

8.3.9 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.

8.3.10 Also See

Bit Set, Bit Test, Bit Clear

8.3.11 Special Applications

8.4 Bit Test

8.4.1 General Form

CC = BITTST (register, bit_position)

CC = ! BITTST (register, bit_position)

8.4.2 Syntax

```
CC = BITTST ( Dreg , uimm5 ) ;          // set CC if bit = 1 (a)

CC = ! BITTST ( Dreg , uimm5 ) ;        // set CC if bit = 0 (a)
```

8.4.3 Syntax Terminology

Dreg: R0, ..., R7

uimm5: 5-bit unsigned field, with a range of 0 through 31

8.4.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

8.4.5 Functional Description

The Bit Test instruction sets or clears the CC bit, based on the bit designated by bit_position in the specified D-register. 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 bit_position range of values is 0 – 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

8.4.6 Flags Affected

This instruction affects flags as follows:

- CC is set if the tested bit is 1; cleared otherwise.

All other flags are unaffected.

8.4.7 Required Mode

User & Supervisor

8.4.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

8.4.9 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.

8.4.10 Also See

Bit Clear, Bit Set, Bit Toggle

8.4.11 Special Applications

8.5 Bit Field Deposit

8.5.1 General Form

dest_reg = DEPOSIT (backgnd_reg, foregnd_reg)

dest_reg = DEPOSIT (backgnd_reg, foregnd_reg) (X)

8.5.2 Syntax

```
Dreg = DEPOSIT ( Dreg, Dreg ) ;           // no extension (b)
Dreg = DEPOSIT ( Dreg, Dreg ) (X) ;       // sign-extended (b)
```

8.5.3 Syntax Terminology

Dreg: R0, ..., R7

8.5.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

8.5.5 Functional Description

The Bit Field Deposit instruction merges the background bit field in backgnd_reg with the foreground bit field in the upper half of foregnd_reg and saves the result into dest_reg. The user determines the length of the foreground bit field and its position in the background field.

Option: Using the (X) syntax, you can sign-extend the deposited bit field. If you specify the sign-extended syntax, the operation does not affect the dest_reg bits that are less significant than the deposited bit field.

The input register bit field definitions appear below.

```
              31.........................................................................................0
backgnd_reg: | bbbb  bbbb  bbbb  bbbb  bbbb  bbbb  bbbb  bbbb  |
              where…
              b = background bit field (32 bits)
```

	31 16	15 8	7 0
foregnd_reg:	nnnn nnnn nnnn nnnn	xxxp pppp	xxxL LLLL

where…

n = foreground bit field (16 bits); the L field determines the actual number of foreground bits used.

p = intended position of foreground bit field LSB in dest_reg (valid range 0 – 31)

L = length of foreground bit field (valid range 0 – 16)

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. See “Example,” below, for more.

Boundary Cases

Consider the following boundary cases:

- Unsigned syntax, L = 0: The architecture copies backgnd_reg contents without modification into dest_reg. By definition, a foreground of zero length is transparent.
- Sign-extended, L = 0 and p = 0: This case loads 0x0000 0000 into dest_reg. The sign of a zero length, zero position foreground is zero; therefore, sign-extended is all zeros.
- Sign-extended, L = 0 and p ≠ 0: The architecture copies the lower order bits of backgnd_reg below position *p* into dest_reg, then sign-extends that number. The foreground value has no effect. For instance, if…

  backgnd_reg = 0x0000 8123,

  L = 0, and

  p = 16,

  then…

  dest_reg = 0xFFFF 8123.

  In this example, the architecture copies bits 15:0 from backgnd_reg into dest_reg, then sign-extends that number.
- Sign-extended, 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 dest_reg.

8.5.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is cleared.
- V is cleared.

All other flags are unaffected.

8.5.7 Required Mode

User & Supervisor

8.5.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

8.5.9 Example

Bit Field Deposit Unsigned

r7 = deposit (r4, r3) ;

IF

	31 0
R4:	1111 1111 1111 1111 1111 1111 1111 1111
	(background bit field)

…and…

	31 16	15 8	7 0
R3:	0000 0000 0000 0**000**	0x7	0x3
	(foreground bit field)	(position)	(length)

Then the BIT FIELD DEPOSIT (unsigned) instruction produces...

	31 7 0
R7:	1111 1111 1111 1111 1111 11**00** **0**111 1111

IF

	31 0
R4:	1111 1111 1111 1111 1111 1111 1111 1111
	(background bit field)

…and…

	31 16	15 8	7 0
R3:	0000 000**0 1111 1010**	0xD	0x9
	(foreground bit field)	(position)	(length)

Then the BIT FIELD DEPOSIT (unsigned) instruction produces...

	31 13 0
R7:	1111 1111 11**01 1111 010**1 1111 1111 1111

Bit Field Deposit Sign-Extended
r7 = deposit (r4, r3) (x) ; // sign-extended

IF

	31 0
R4:	1111 1111 1111 1111 1111 1111 1111 1111
	(background bit field)

…and…

	31 16	15 8	7 0
R3:	0101 1010 0101 1**010**	0x7	0x3
	(foreground bit field)	(position)	(length)

Then the BIT FIELD DEPOSIT (sign-extended) instruction produces...

	31 7 0
R7:	**0000 0000 0000 0000 0000 0001 0**111 1111

IF

	31 0
R4:	1111 1111 1111 1111 1111 1111 1111 1111
	(background bit field)

…and…

	31 16	15 8	7 0
R3:	0000 100**1 1010 1100**	0xD	0x9
	(foreground bit field)	(position)	(length)

Then the BIT FIELD DEPOSIT (sign-extended) instruction produces...

```
      31                        13                  0
R7:   1111 1111 1111 0101 1001 1111 1111 1111
```

8.5.10 Also See

Bit Field Extraction

8.5.11 Special Applications

Video image overlay algorithms.

8.6 Bit Field Extraction

8.6.1 General Form

dest_reg = EXTRACT (scene_reg, pattern_reg) (Z)

dest_reg = EXTRACT (scene_reg, pattern_reg) (X)

8.6.2 Syntax

```
Dreg = EXTRACT ( Dreg, Dreg_lo ) (Z) ;        // zero-extended (b)
Dreg = EXTRACT ( Dreg, Dreg_lo ) (X) ;        // sign-extended (b)
```

8.6.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo: R0.L, ..., R7.L

8.6.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

8.6.5 Functional Description

The Bit Field Extraction instruction moves only specific bits from the scene_reg into the low-order bits of the dest_reg. The user determines the length of the pattern bit field and its position in the scene field.

The user has the choice of using the (X) syntax to perform sign-extend extraction or the (Z) syntax to perform zero-extend extraction.

The input register bit field definitions appear below.

	31 0
scene_reg:	ssss ssss ssss ssss ssss ssss ssss ssss

where…

s = scene bit field (32 bits)

	15 8	7 0
pattern_reg:	xxxx pppp	xxxx LLLL

where…

p = position of pattern bit field LSB in scene_reg
(valid range 0 – 31)

L = length of pattern bit field (valid range 0 – 31)

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.

Boundary Case

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 scene_reg 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 scene_reg 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 dest_reg.

8.6.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is cleared.
- V is cleared.

All other flags are unaffected.

8.6.7 Required Mode

User & Supervisor

8.6.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

8.6.9 Example

Bit Field Extraction Unsigned

```
r7  = extract (r4, r3.l) (z);      // zero-extended
```

IF	31 7 0
R4:	1010 0101 1010 0101 1100 0**011** **1**010 1010
	(scene bit field)

…and…	31 16	15 8	7 0
R3:	xxxx xxxx xxxx xxxx	0x7	0x4
		(position)	(length)

Then the Bit Field Extraction (unsigned) instruction produces...	31 0
R7:	0000 0000 0000 0000 0000 0000 0000 **0111**

IF	31 13 0
R4:	1010 0101 10**10** **0101** **110**0 0011 1010 1010
	(scene bit field)

…and…	31 16	15 8	7 0
R3:	xxxx xxxx xxxx xxxx	0xD	0x9

(position) (length)

Then the Bit Field Extraction (unsigned) instruction produces...

31 0

R7: 0000 0000 0000 0000 0000 000**1** **0010** **1110**

Bit Field Extraction Sign-Extended

```
r7  = extract (r4, r3.l) (x) ;      // sign-extended
```

IF

31 7 0

R4: 1010 0101 1010 0101 1100 0**011** **1**010 1010

(scene bit field)

…and…

	31 16	15 8	7 0
R3:	xxxx xxxx xxxx xxxx	0x7	0x4
		(position)	(length)

Then the Bit Field Extraction (sign-extended) instruction produces...

31 0

R7: 0000 0000 0000 0000 0000 0000 0000 0111

IF

31 13 0

R4: 1010 0101 10**10** **0101** **110**0 0011 1010 1010

(scene bit field)

…and…

	31 16	15 8	7 0
R3:	xxxx xxxx xxxx xxxx	0xD	0x9
		(position)	(length)

Then the Bit Field Extraction (sign-extended) instruction produces...

31 0

R7: 1111 1111 1111 1111 1111 111**1 0010 1110**

8.6.10 Also See

Bit Field Deposit

8.6.11 Special Applications

Video image pattern recognition and separation algorithms.

8.7 Bit Multiplex

8.7.1 General Form

BITMUX (source_1, source_0, A0) (ASR)

BITMUX (source_1, source_0, A0) (ASL)

8.7.2 Syntax

```
BITMUX ( Dreg , Dreg , A0 ) (ASR) ;       // shift right, LSB is shifted out (b)
BITMUX ( Dreg , Dreg , A0 ) (ASL) ;       // shift left, MSB is shifted out (b)
```

8.7.3 Syntax Terminology

Dreg: R0, ..., R7

8.7.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

8.7.5 Functional Description

The Bit Multiplex instruction merges bit streams. The instruction has two versions, Shift Right and Shift Left. This instruction overwrites the contents of source_1 and source_0.

In the Shift Right version, the processor performs the following sequence.

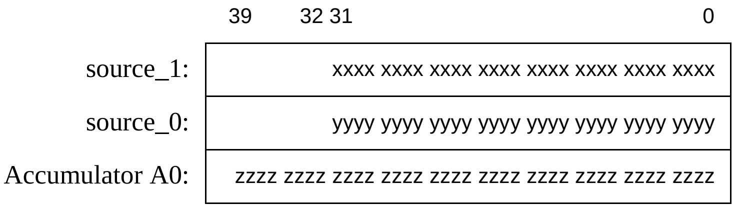
1. Right shift Accumulator A0 by one bit. Right shift the LSB of source_1 into the MSB of the Accumulator.
2. Right shift Accumulator A0 by one bit. Right shift the LSB of source_0 into the MSB of the Accumulator.

In the Shift Left version, the processor performs the following sequence.

1. Left shift Accumulator A0 by one bit. Left shift the MSB of source_0 into the LSB of the Accumulator.
2. Left shift Accumulator A0 by one bit. Left shift the MSB of source_1 into the LSB of the Accumulator.

Source_1 and source_0 must not be the same D-register.

IF


39
32 31
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

a SHIFT RIGHT instruction produces…	39 32 31 0
source_1:	0xxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx (shifted right 1 place)
source_0:	0yyy yyyy yyyy yyyy yyyy yyyy yyyy yyyy (shifted right 1 place)
Accumulator A0:	**yx**zz zzzz zzzz zzzz zzzz zzzz zzzz zzzz zzzz zzzz (shifted right 2 places)

a SHIFT LEFT instruction produces…	39 32 31 0
source_1:	xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxx0 (shifted left 1 place)
source_0:	yyyy yyyy yyyy yyyy yyyy yyyy yyyy yyy0 (shifted left 1 place)
Accumulator A0:	zzzz zzzz zzzz zzzz zzzz zzzz zzzz zzzz zzzz zz**yx** (shifted left 2 places)

8.7.6 Flags Affected

None

8.7.7 Required Mode

User & Supervisor

8.7.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

8.7.9 Example

bitmux (r2, r3, a0) (asr) ; // right shift

IF	31 0
R2:	`1010 0101 1010 0101 1100 0011 1010 1010`

…and…

31 0

R3: 1100 0011 1010 1010 1010 0101 1010 0101

…and…

39 0

A0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0111

then the SHIFT RIGHT instruction produces…

31 0

R2: **0**101 0010 1101 0010 1110 0001 1101 0101

…and…

31 0

R3: **0**110 0001 1101 0101 0101 0010 1101 0010

…and…

39 0

A0: **10**00 0000 0000 0000 0000 0000 0000 0000 0000 0001

bitmux (r3, r2, a0) (asl) ; // left shift

IF

31 0

R3: 1010 0101 1010 0101 1100 0011 1010 1010

…and…

31 0

R2: 1100 0011 1010 1010 1010 0101 1010 0101

…and…

39 0

A0: 1110 0000 0000 0000 0000 0000 0000 0000 0000 0111 then the SHIFT LEFT instruction produces…

31 0

R3: 0100 1011 0100 1011 1000 0111 0101 010**0**

…and…

31 0

R2: 1000 0111 0101 0101 0100 1011 0100 101**0**

…and…

39 0

A0: 1000 0000 0000 0000 0000 0000 0000 0000 0001 11**11**

8.7.10 Also See

8.7.11 Special Applications

Use the Bit Multiplex instruction for convolutional encoder algorithms.

8.8 Ones Population Count

8.8.1 General Form

dest_reg = ONES src_reg

8.8.2 Syntax

```
Dreg_lo = ONES Dreg ;          // (b)
```

8.8.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo: R0.L, ..., R7.L

8.8.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

8.8.5 Functional Description

The Ones Population Count instruction loads the number of 1’s contained in the src_reg into the lower half of the dest_reg.

The range of possible values loaded into dest_reg is 0 – 32.

The dest_reg and src_reg can be the same D-register. Otherwise, the Ones Population Count instruction does not modify the contents of src_reg.

8.8.6 Flags Affected

None

8.8.7 Required Mode

User & Supervisor

8.8.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

8.8.9 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.

8.8.10 Also See

8.8.11 Special Applications

This instruction can be used for software parity testing.

SHIFT / ROTATE OPERATIONS 9

Instruction Summary

This chapter discusses the instructions that manipulate bit operations. Users can take advantage of these instructions to perform logical and arithmetic shifts, combine addition operations with shifts, and rotate a registered number through the CC bit.

9.1 Add with Shift

9.1.1 General Form

dest_pntr = (dest_pntr + src_reg) << 1

dest_pntr = (dest_pntr + src_reg) << 2

dest_reg = (dest_reg + src_reg) << 1

dest_reg = (dest_reg + src_reg) << 2

9.1.2 Syntax

POINTER OPERATIONS

```
Preg = ( Preg + Preg ) << 1 ;     // dest_reg = (dest_reg + src_reg) x 2   (a)
Preg = ( Preg + Preg ) << 2 ;     // dest_reg = (dest_reg + src_reg) x 4   (a)
```

DATA OPERATIONS

```
Dreg = (Dreg + Dreg) << 1 ;       // dest_reg = (dest_reg + src_reg) x 2   (a)
Dreg = (Dreg + Dreg) << 2 ;       // dest_reg = (dest_reg + src_reg) x 4   (a)
```

9.1.3 Syntax Terminology

Preg: P0, ..., P5

Dreg: R0, ..., R7

9.1.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

9.1.5 Functional Description

The Add with Shift instruction combines an addition operation with a one- or two-place logical shift left. Of course, a left shift accomplishes a x2 multiplication on sign-extended numbers. Saturation is not supported.

The Add with Shift instruction does not intrinsically modify values that are strictly input. However, dest_reg serves as an input as well as the result, so dest_reg is intrinsically modified.

9.1.6 Flags Affected

The D-register versions of this instruction affects flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- V is set if result overflows; cleared if no overflow.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

The P-register versions of this instruction do not affect any flags.

9.1.7 Required Mode

User & Supervisor

9.1.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

9.1.9 Example

```
p3 = (p3+p2)<<1 ;                    // p3 = (p3 + p2) * 2
p3 = (p3+p2)<<2 ;                    // p3 = (p3 + p2) * 4
r3 = (r3+r2)<<1 ;                    // r3 = (r3 + r2) * 2
r3 = (r3+r2)<<2 ;                    // r3 = (r3 + r2) * 4
```

9.1.10 Also See

Shift with Add, Multiply (Modulo 2^{32}), Logical Shift, Arithmetic Shift, Add

9.1.11 Special Applications

9.2 Shift with Add

9.2.1 General Form

dest_pntr = adder_pntr + (src_pntr << 1)

dest_pntr = adder_pntr + (src_pntr << 2)

9.2.2 Syntax

```
Preg = Preg + ( Preg << 1 ) ;     // adder_pntr + (src_pntr x 2)   (a)
Preg = Preg + ( Preg << 2 ) ;     // adder_pntr + (src_pntr x 4)   (a)
```

9.2.3 Syntax Terminology

Preg: P0, ..., P5

9.2.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

9.2.5 Functional Description

The Shift with Add instruction combines a one- or two-place 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.

9.2.6 Flags Affected

None

9.2.7 Required Mode

User & Supervisor

9.2.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

9.2.9 Example

```
p3 = p0+(p3<<1) ;               // p3 = (p3 * 2) + p0
p3 = p0+(p3<<2) ;               // p3 = (p3 * 4) + p0
```

9.2.10 Also See

Add with Shift, Multiply (Modulo 2^{32}), Logical Shift, Arithmetic Shift, Add

9.2.11 Special Applications

9.3 Arithmetic Shift

9.3.1 General Form

dest_reg >>>= shift_magnitude

dest_reg = src_reg >>> shift_magnitude (opt_sat)

dest_reg = src_reg << shift_magnitude (S)

accumulator = accumulator >>> shift_magnitude

dest_reg = ASHIFT src_reg BY shift_magnitude (opt_sat)

accumulator = ASHIFT accumulator BY shift_magnitude

9.3.2 Syntax

CONSTANT SHIFT MAGNITUDE

Dreg >>>= uimm5 ;	// arithmetic right shift (a)
Dreg <<= uimm5 ;	// *logical* left shift (a)
Dreg_lo_hi = Dreg_lo_hi >>> uimm4 ;	// arithmetic right shift (b)
Dreg_lo_hi = Dreg_lo_hi << uimm4 (S) ;	// arithmetic left shift (b)
Dreg = Dreg >>> uimm5 ;	// arithmetic right shift b)
Dreg = Dreg << uimm5 (S) ;	// arithmetic left shift (b)
A0 = A0 >>> uimm5 ;	// arithmetic right shift (b)
A0 = A0 << uimm5 ;	// *logical* left shift (b)
A1 = A1 >>> uimm5 ;	// arithmetic right shift (b)
A1 = A1 << uimm5 ;	// *logical* left shift (b)

REGISTERED SHIFT MAGNITUDE

Dreg >>>= Dreg ;	// arithmetic right shift (a)
Dreg <<= Dreg ;	// *logical* left shift (a)
Dreg_lo_hi = ASHIFT Dreg_lo_hi BY Dreg_lo (opt_sat) ;	// arithmetic right shift (b)
Dreg = ASHIFT Dreg BY Dreg_lo (opt_sat) ;	// arithmetic right shift (b)
A0 = ASHIFT A0 BY Dreg_lo ;	// arithmetic right shift (b)
A1 = ASHIFT A1 BY Dreg_lo ;	// arithmetic right shift (b)

9.3.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

Dreg_lo: R0.L, ..., R7.L

uimm4: 4-bit unsigned field, with a range of 0 through 15

uimm5: 5-bit unsigned field, with a range of 0 through 31

opt_sat: optional “(S)” (without the quotes) to invoke saturation of the result. Not optional on versions that show “(S)” in the syntax.

9.3.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

9.3.5 Functional Description

The Arithmetic Shift instruction shifts a registered number a specified distance and direction while preserving the sign of the original number. The sign bit value back-fills the left-most bit positions vacated by the arithmetic right shift.

Specific versions of arithmetic left shift are supported, too. Arithmetic left shift saturates the result if the value is shifted too far. A left shift that would otherwise lose non-sign bits off the left-hand side saturates to the maximum positive or negative value instead.

The “ASHIFT” versions of this instruction support two modes:

1. default -- arithmetic right shifts and logical left shifts. Logical left shifts do not guarantee sign bit preservation. The “ASHIFT” versions automatically select arithmetic and logical shift modes based on the sign of the shift_magnitude.
2. saturation mode -- arithmetic right and left shifts that saturate if the value is shifted left too far.

The “>>>=” and “>>>” versions of this instruction supports only arithmetic right shifts. If left shifts are desired, the programmer must explicitly use arithmetic “<<“ (saturating) or logical “<<“ (non-saturating) instructions.

Note: Logical left shift instructions are duplicated in the Syntax section for programmer convenience. See the Logical Shift instruction for details on those operations.

The Arithmetic Shift instruction supports 16-bit and 32-bit instruction length.

- The “>>>=” syntax instruction is 16-bits in length, allowing for smaller code at the expense of flexibility.
- The “>>>”, “<<“, and “ASHIFT” syntax instructions are 32-bits in length, providing a separate source and destination register, alternative data sizes, and parallel issue with Load/ Store instructions.

Both syntaxes support constant and registered shift magnitudes.

Table 9-1. Arithmetic Shifts

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). Only right shifting is supported in this syntax; there is no equivalent “<<<=” arithmetic left shift syntax. However, logical left shift is supported. See the Logical Shift instruction.
“>>>”, “<<“, 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 and 40-bit Accumulator registers by up to -32 – 31 places; the distance is determined by the lower 6 bits (sign extended) of the shift_magnitude.

For the ASHIFT versions, the sign of the shift magnitude determines the direction of the shift.

- Positive shift magnitudes produce LOGICAL LEFT shifts.
- Negative shift magnitudes produce ARITHMETIC RIGHT shifts.

In essence, the magnitude is the power of 2 multiplied by the src_reg number. Positive magnitudes cause multiplication (N x 2^n) whereas negative magnitudes produce division (N x 2^{-n} or N / 2^n).

The dest_reg and src_reg can be a 16-, 32-, or 40-bit register. Some versions of the Arithmetic Shift instruction support optional saturation.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

For 16-bit src_reg, valid shift magnitudes are –16 through +15, zero included. For 32- and 40-bit src_reg, valid shift magnitudes are –32 through +31, zero included. The Arithmetic Shift instruction masks and ignores bits that are more significant than those allowed.

The D-register versions of this instruction shift 16 or 32 bits for half-word and word registers, respectively. The Accumulator versions shift all 40 bits of those registers.

The D-register versions of this instruction do not implicitly modify the src_reg values. Optionally, dest_reg can be the same D-register as src_reg. So doing explicitly modifies the source register.

The Accumulator versions always modify the Accumulator source value.

9.3.6 Options

Option (S) invokes saturation of the result.

In the default case – without the saturation option – numbers can be left-shifted so far that all the sign bits overflow and are lost. However, when the saturation option is enabled, a left shift that would otherwise shift non-sign bits off the left-hand side saturates to the maximum positive or negative value instead. Consequently, with saturation enabled, the result always keeps the same sign as the original number.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

9.3.7 Flags Affected

The versions of this instruction that send results to a Dreg set flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- V is set if result overflows; cleared if no overflow.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

The versions of this instruction that send results to a Accumulator A0 set flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AV0 is set if result is zero; cleared if non-zero.
- AV0S is set if AV0 is set; unaffected otherwise.

All other flags are unaffected.

The versions of this instruction that send results to a Accumulator A1 set flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AV1 is set if result is zero; cleared if non-zero.
- AV1S is set if AV1 is set; unaffected otherwise.

All other flags are unaffected.

9.3.8 Required Mode

User & Supervisor

9.3.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

9.3.10 Example

```
r0 >>>= 19  ;                                    /* 16-bit instruction length arithmetic
                                                 right shift */
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
r4 = r2 >>> 20 ;                                 // arithmetic right shift, word
A0 = A0 >>> 1 ;                                  // arithmetic right shift, Accumulator
r0 >>>= r2 ;                                     /* 16-bit instruction length arithmetic
                                                 right shift */

r3.l = r0.h << 12 (S) ;                          // arithmetic left shift
r5 = r2 << 24(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) ;
r4 = ashift r2 by r7.l ;                         // shift, word
r4 = ashift r2 by r7.l (s) ;                     // shift, word, saturated
A0 = ashift A0 by r7.l ;                         // shift, Accumulator
A1 = ashift A1 by r7.l ;                         // shift, Accumulator
                                                // If r0.h = -64, then performing . . .
r3.h = r0.h >>> 4 ;                             // . . . produces r3.h = -4, preserving the sign
```

9.3.11 Also See

Vector Arithmetic Shift, Vector Logical Shift, Logical Shift, Shift with Add, Rotate

9.3.12 Special Applications

Typically, use Arithmetic shifting to multiply, divide, and normalize signed numbers.

9.4 Logical Shift

9.4.1 General Form

dest_pntr = src_pntr >> 1

dest_pntr = src_pntr >> 2

dest_pntr = src_pntr << 1

dest_pntr = src_pntr << 2

dest_reg >>= shift_magnitude

dest_reg <<= shift_magnitude

dest_reg = src_reg >> shift_magnitude

dest_reg = src_reg << shift_magnitude

dest_reg = LSHIFT src_reg BY shift_magnitude

9.4.2 Syntax

POINTER SHIFT, FIXED MAGNITUDE

```
Preg = Preg >> 1 ;                      // right shift by 1 bit (a)
Preg = Preg >> 2 ;                      // right shift by 2 bit (a)
Preg = Preg << 1 ;                      // left shift by 1 bit (a)
Preg = Preg << 2 ;                      // left shift by 2 bit (a)
```

DATA SHIFT, CONSTANT SHIFT MAGNITUDE

```
Dreg >>= uimm5 ;                        // right shift (a)
Dreg <<= uimm5 ;                        // left shift (a)
Dreg_lo_hi = Dreg_lo_hi >> uimm4 ;      // right shift (b)
Dreg_lo_hi = Dreg_lo_hi << uimm4 ;      // left shift (b)
Dreg = Dreg >> uimm5  ;                 // right shift (b)
Dreg = Dreg << uimm5 ;                  // left shift (b)
A0 = A0 >> uimm5  ;                     // right shift (b)
A0 = A0 << uimm5 ;                     // left shift (b)
A1 = A1 << uimm5  ;                     // left shift (b)
```

```
A1 = A1 >> uimm5 ;                                // right shift (b)
```

DATA SHIFT, REGISTERED SHIFT MAGNITUDE

```
Dreg >>= Dreg ;                                   // right shift (a)

Dreg <<= Dreg ;                                   // left shift (a)

Dreg_lo_hi = LSHIFT Dreg_lo_hi BY Dreg_lo ;       // (b)

Dreg = LSHIFT Dreg BY Dreg_lo ;                   // (b)

A0 = LSHIFT A0 BY Dreg_lo ;                       // (b)

A1 = LSHIFT A1 BY Dreg_lo  ;                      // (b)
```

9.4.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo: R0.L, ..., R7.L

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

Preg: P0, ..., P5

uimm4: 4-bit unsigned field, with a range of 0 through 15

uimm5: 5-bit unsigned field, with a range of 0 through 31

9.4.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

9.4.5 Functional Description

The Logical Shift instruction logically shifts a registered number a specified distance and direction.

Logical shifts discard any bits shifted out of the register and backfill vacated bits with zeros.

Four versions of the Logical Shift instruction support pointer shifting. The instruction does not implicitly modify the input src_pntr value. For the P-register versions of this instruction, dest_pntr can be the same P-register as src_pntr. Doing so explicitly modifies the source register.

The rest of this description applies to the data shift versions of this instruction relating to D-registers and Accumulators.

The Logical Shift instruction supports 16-bit and 32-bit instruction length.

- The “>>=” and “<<=” syntax instruction is 16-bits in length, allowing for smaller code at the expense of flexibility.

- The “>>”, “<<”, and “LSHIFT” syntax instruction is 32-bits in length, providing a separate source and destination register, alternative data sizes, and parallel issue with Load/Store instructions.

Both syntaxes support constant and registered shift magnitudes.

Table 9-2. Logical Shifts

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 Dreg and 40-bit Accumulator registers by up to -32 – 31 places; the distance is determined by the lower 6 bits (sign extended) of the shift_magnitude.

For the LSHIFT version, the sign of the shift magnitude determines the direction of the shift.

- Positive shift magnitudes produce LEFT shifts.
- Negative shift magnitudes produce RIGHT shifts.

The dest_reg and src_reg can be a 16-, 32-, or 40-bit register.

For the LSHIFT instruction, the shift magnitude is the lower 6 bits of the Dreg_lo, sign extended. The Dreg >>= Dreg and Dreg <<= Dreg instructions use the entire 32-bits of magnitude.

The D-register versions of this instruction shift 16 or 32 bits for half-word and word registers, respectively. The Accumulator versions shift all 40 bits of those registers.

40-bit Accumulator values can be shifted by up to -32 to +31 bit places.

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 0x00000000.

A shift magnitude of zero performs no shift operation at all.

The D-register versions of this instruction do not implicitly modify the src_reg values. Optionally, dest_reg can be the same D-register as src_reg. Doing so explicitly modifies the source register.

9.4.6 Flags Affected

The P-register versions of this instruction do not affect any flags.

The versions of this instruction that send results to a Dreg set flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- V is cleared.

All other flags are unaffected.

The versions of this instruction that send results to a Accumulator A0 set flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AV0 is cleared.

All other flags are unaffected.

The versions of this instruction that send results to a Accumulator A1 set flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AV1 is cleared.

All other flags are unaffected.

9.4.7 Required Mode

User & Supervisor

9.4.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

9.4.9 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
r3 >>= 17 ;                  // data right shift
r3 <<= 17 ;                  // data left shift
r3.l = r0.l >> 4 ;           // data right shift, half-word register
r3.l = r0.h >> 4 ;           /* same as above; half-word register
                             combinations are arbitrary */
r3.h = r0.l << 12 ;          // data left shift, half-word register
r3.h = r0.h << 14;           /* same as above; half-word register
                             combinations are arbitrary */
r3 = r6 >> 4  ;              // right shift, 32-bit word
r3 = r6 << 4  ;              // left shift, 32-bit word
a0 = a0 >> 7  ;              // Accumulator right shift
a1 = a1 >> 25  ;             // Accumulator right shift
a0 = a0 << 7  ;              // Accumulator left shift
a1 = a1 << 14  ;             // Accumulator left shift
r3 >>= r0 ;                  // data right shift
r3 <<= r1 ;                  // data left shift
```

```
r3.l = lshift r0.l by r2.l ;                    // shift direction controlled by sign of R2.L
r3.h = lshift r0.l by  r2.l ;
a0 = lshift a0 by r7.l ;
a1 = lshift  a1 by r7.l ;
                                               // If r0.h = -64 (or 0xFFC0), then performing . . .
r3.h = r0.h >> 4 ;                             /* . . . produces r3.h = 0x0FFC (or 4092), losing
                                               the sign */
```

9.4.10 Also See

Arithmetic Shift, Rotate, Shift with Add, Vector Arithmetic Shift, Vector Logical Shift

9.4.11 Special Applications

9.5 Rotate

9.5.1 General Form

dest_reg = ROT src_reg BY rotate_magnitude

accumulator_new = ROT accumulator_old BY rotate_magnitude

9.5.2 Syntax

CONSTANT ROTATE MAGNITUDE

Dreg = ROT Dreg BY imm6 ;   // (b)

A0 = ROT A0 BY imm6 ;   // (b)

A1 = ROT A1 BY imm6 ;   // (b)

REGISTERED ROTATE MAGNITUDE

Dreg = ROT Dreg BY Dreg_lo ;   // (b)

A0 = ROT A0 BY Dreg_lo ;   // (b)

A1 = ROT A1 BY Dreg_lo ;   // (b)

9.5.3 Syntax Terminology

Dreg: R0, ..., R7

imm6: 6-bit signed field, with a range of -32 through 31

9.5.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

9.5.5 Functional Description

The Rotate instruction rotates a registered number 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.

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.

Unlike shift operations, Rotate 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.

The D-register versions of this instruction rotate all 32 bits. The Accumulator versions rotate all 40 bits of those registers.

The D-register versions of this instruction do not implicitly modify the src_reg values. Optionally, dest_reg can be the same D-register as src_reg. Doing so explicitly modifies the source register.

9.5.6 Flags Affected

The following flags are affected by the Rotate instruction:

- CC contains the latest value shifted into it.

All other flags are unaffected.

9.5.7 Required Mode

User & Supervisor

9.5.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

9.5.9 Example

```
r4 = rot r1 by 8  ;                        // rotate left
r4 = rot r1 by -5 ;                        // rotate right
a0 = rot  a0 by 22  ;                      // rotate Accumulator left
a1 = rot  a1 by -31  ;                     // rotate Accumulator right
r4 = rot r1 by r2.l ;
a0 = rot  a0 by r3.l ;
a1 = rot  a1 by r7.l ;
```

9.5.10 Also See

Arithmetic Shift, Logical Shift

9.5.11 Special Applications

ARITHMETIC OPERATIONS 10

Instruction Summary

This chapter discusses the instructions that specify arithmetic operations. Users can take advantage of these instructions to add, subtract, divide, and multiply, calculate and store absolute values, detect exponents, round, saturate, and return the number of sign bits.

10.1 Absolute Value

10.1.1 General Form

dest_reg = ABS src_reg

10.1.2 Syntax

```
A0 = ABS A0 ;                       // (b)
A0 = ABS A1 ;                       // (b)
A1 = ABS A0 ;                       // (b)
A1 = ABS A1 ;                       // (b)
A1 = ABS A1, A0 = ABS A0 ;          // (b)
Dreg = ABS Dreg ;                   // (b)
```

10.1.3 Syntax Terminology

Dreg: R0, ..., R7

10.1.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.1.5 Functional Description

The Absolute Value instruction calculates the absolute value of a 32-bit register and stores it into a 32-bit dest_reg 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.

The ABS operation can also be performed on both Accumulators by a single instruction.

10.1.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if result is zero; cleared if non-zero. In the case of two simultaneous operations, AZ represents the logical “OR” of the two.
- AN is cleared.
- V is set if the maximum negative value is saturated to the maximum positive value and the dest_reg is a Dreg; cleared if no saturation.

- VS is set if V is set; unaffected otherwise.
- AV0 is set if result overflows and the dest_reg is A0; cleared if no overflow.
- AV0S is set if AV0 is set; unaffected otherwise.
- AV1 is set if result overflows and the dest_reg is A1; cleared if no overflow.
- AV1S is set if AV1 is set; unaffected otherwise.

All other flags are unaffected.

10.1.7 Required Mode

User & Supervisor

10.1.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.1.9 Example

```
a0 = abs a0 ;
a0 = abs a1 ;
a1 = abs a0 ;
a1 = abs a1 ;
a1 = abs a1, a0=abs a0 ;
r3 = abs r1 ;
```

10.1.10 Also See

Vector Absolute Value (in “Vector Operations” chapter)

10.1.11 Special Applications

None

10.2 Add

10.2.1 General Form

dest_reg = src_reg_1 + src_reg_2

10.2.2 Syntax

POINTER REGISTER ADDITION

```
Preg = Preg + Preg ;                    // (a)
```

32-BIT OPERANDS, 32-BIT RESULT

```
Dreg = Dreg + Dreg  ;                   /* no saturation support but
                                        shorter instruction length (a) */

Dreg = Dreg + Dreg  (sat_flag) ;       /* saturation optionally supported, but at the cost of longer
                                       instruction length (b) */
```

16-BIT OPERANDS, 16-BIT RESULT

```
Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi  (sat_flag) ; // (b)
```

10.2.3 Syntax Terminology

Preg: P0, ..., P5, SP, FP

Dreg: R0, ..., R7

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

sat_flag: non-optional saturation flag, (S) or (NS)

10.2.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

10.2.5 Functional Description

The Add instruction adds two source values and places the result in a destination register.

There are two ways to specify addition on 32-bit data in D-registers. One that is 16-bit instruction length does not support saturation. The other instruction, which is 32-bit instruction length, optionally supports saturation. The larger DSP instruction can sometimes save execution time because it can be issued in parallel with certain other instructions. See “Parallel Issue”.

The instructions for 16-bit data use half-word data register operands and store the result in a half-word data register.

All instructions for 16-bit data are 32-bit instruction length.

Option: In the syntax, where sat_flag appears, substitute one of the following values:

- (S) – saturate the result
- (NS) – no saturation

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

10.2.6 Flags Affected

D-register versions of this instruction set flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is set if the operation generates a carry; cleared if no carry.
- V is set if result overflows; cleared if no overflow.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

The P-register versions of this instruction do not affect any flags.

10.2.7 Required Mode

User & Supervisor

10.2.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

10.2.9 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
p5 = p3 + p0 ;

                                 // 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) ;
```

10.2.10 Also See

Modify – Increment, Round – 12-bit, Round – 20-bit, Shift with Add and Add with Shift (both in “Shift/Rotate Operations” chapter), Vector Add/Subtract (in “Vector Operations” chapter)

10.2.11 Special Applications

None

10.3 Add Immediate

10.3.1 General Form

register += constant

10.3.2 Syntax

```
Dreg += imm7 ;          // Dreg = Dreg + constant (a)
Preg += imm7 ;          // Preg = Preg + constant (a)
Ireg += 2 ;             /* increment Ireg by 2, half-word address pointer
                        increment (a)*/
Ireg += 4 ;             // word address pointer increment (a)
```

10.3.3 Syntax Terminology

Dreg: R0, ..., R7

Preg: P0, ..., P5, SP, FP

Ireg: I0, ..., I3

imm7: 7-bit signed field, with the range of -64 through 63

10.3.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

10.3.5 Functional Description

The Add Immediate instruction adds a constant value to a register without saturation.

Note: To subtract immediate values from I-registers, use the Subtract Immediate instruction.

10.3.6 Flags Affected

D-register versions of this instruction set flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is set if the operation generates a carry; cleared if no carry.
- V is set if result overflows; cleared if no overflow.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

The P-register and I-register versions of this instruction do not affect any flags.

10.3.7 Required Mode

User & Supervisor

10.3.8 Parallel Issue

The Data Register and Pointer Register versions of the Add Immediate instruction cannot be issued in parallel with other instructions.

The Index Register versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, "Issuing Parallel Instructions."

The Data Register and Pointer Register versions of this instruction cannot be issued in parallel with other instructions.

10.3.9 Example

```
r0 += 40 ;
p5 += -4  ;                              // decrement by adding a negative value
i0 += 2 ;
i1 += 4 ;
```

10.3.10 Also See

Subtract Immediate

10.3.11 Special Applications

None

10.4 Divide Primitive

10.4.1 General Form

DIVS (dividend_register, divisor_register)

DIVQ (dividend_register, divisor_register)

10.4.2 Syntax

DIVS (Dreg, Dreg);	/* Initialize for DIVQ. Set the AQ flag based on the signs of the 32-bit dividend and the 16-bit divisor. Left shift the dividend one bit. Copy AQ into the dividend LSB. (a) */
DIVQ (Dreg, Dreg);	/* Based on AQ flag, either add or subtract the divisor from the dividend. Then set the AQ flag based on the MSB's of the 32-bit dividend and the 16-bit divisor. Left shift the dividend one bit. Copy the logical inverse of AQ into the dividend LSB. (a) */

10.4.3 Syntax Terminology

Dreg: R0, ..., R7

10.4.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

10.4.5 Functional Description

The Divide Primitive instruction versions are the foundation elements of a non-restoring conditional add-subtract division algorithm. See “Example” 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 flag 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 flag, 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 flag 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 flag, then re-initializes the AQ flag and dividend for the next iteration. If AQ is 1, addition is performed; if AQ is 0, then subtraction.

See “Flags Affected”, below, for the conditions that set and clear the AQ flag.

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 binary point, and
- NR represent the number of bits to the right of the binary point of the dividend (numerator);
- DL represent the number of bits to the left of the binary point, and
- DR represent the number of bits to the right of the binary point of the divisor (denominator).

Then the quotient has NL – DL + 1 bits to the left of the binary point and NR – DR – 1 bits to the right of the binary point. See the example illustration, below.

Dividend	BBBB B .	BBB BBBB BBBB BBBB BBBB BBBB BBBB
(numerator)	NL bits	NR bits
Divisor	BB .	BB BBBB BBBB BBBB
(denominator)	DL bits	DR bits
Quotient	BBBB .	BBBB BBBB BBBB
	NL - DL +1	NR - DR - 1
	(5 - 2 + 1)	(27 - 14 - 1)
	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, re-scale 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 useable 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).

Error Conditions:

Two special cases can produce invalid or inaccurate results. Software can trap and correct both cases.

1. 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, use the following solution:

    - Before performing the division, save the sign of the divisor in a scratch register.
    - Calculate the absolute value of the divisor and use that value as the divisor operand in the Divide Primitive instructions.
    - After the divide sequence concludes, multiply the resulting quotient by the original divisor sign.

    The quotient then has the correct magnitude and sign.

2. The Divide Primitive instructions do not support unsigned division by a divisor greater than 0x7FFF. If such divisions are necessary, pre-scale 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, pre-scaling 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 solution:

    - Save the original (un-scaled) dividend and divisor in scratch registers.
    - Pre-scale both operands as prescribed and perform the division as usual.
    - Multiply the resulting quotient by the un-scaled divisor. Do not corrupt the quotient by the multiplication step.
    - Subtract the product from the un-scaled dividend. This step produces an error value.
    - Compare the error value to the un-scaled divisor.
        - — If error > divisor, add one LSB to the quotient.
        - — If error < divisor, subtract one LSB from the quotient.
        - — If error = divisor, do nothing.

Tested examples of these solutions are planned to be added in a later edition of this document.

10.4.6 Flags Affected

This instruction affects flags as follows:

- AQ equals dividend_MSB Exclusive-OR divisor_MSB

where dividend is a 32-bit value and
divisor is a 16-bit value.

All other flags are unaffected.

10.4.7 Required Mode

User & Supervisor

10.4.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

10.4.9 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
                                 flag 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 32
                                 bits.*/
/* r0 contains the quotient (70/5 = 14). */
```

10.4.10 Also See

Multiply (Modulo 2^{32}), Zero Overhead Loop

10.4.11 Special Applications

None

10.5 Exponent Detection

10.5.1 General Form

dest_reg = EXPADJ (sample_register, exponent_register)

10.5.2 Syntax

```
Dreg_lo = EXPADJ ( Dreg, Dreg_lo ) ;              // 32-bit sample (b)
Dreg_lo = EXPADJ ( Dreg_lo_hi, Dreg_lo ) ;        // one 16-bit sample (b)
Dreg_lo = EXPADJ ( Dreg, Dreg_lo ) (V) ;          // two 16-bit samples (b)
```

10.5.3 Syntax Terminology

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

Dreg_lo: R0.L, ..., R7.L

Dreg: R0, ..., R7

10.5.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.5.5 Functional 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 D-register. So doing explicitly modifies the exponent_register.

The valid range of exponents is 0 – 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.

10.5.6 Flags Affected

None

10.5.7 Required Mode

User & Supervisor

10.5.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.5.9 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.

10.5.10 Also See

Sign Bit for more information about exponents

10.5.11 Special Applications

EXPADJ detects 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.

10.6 Maximum

10.6.1 General Form

dest_reg = MAX (src_reg_0, src_reg_1)

10.6.2 Syntax

```
Dreg = MAX ( Dreg , Dreg ) ;        // 32-bit operands (b)
```

10.6.3 Syntax Terminology

Dreg: R0, ..., R7

10.6.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.6.5 Functional Description

The Maximum instruction returns the maximum, or most positive, value of the source registers. The operation subtracts src_reg_1 from src_reg_0 and selects the output based on the signs of the input values and the arithmetic flags.

The Maximum instruction does not implicitly modify input values. The dest_reg can be the same D-register as one of the source registers. Doing so explicitly modifies the source register.

10.6.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- V is cleared.

All other flags are unaffected.

10.6.7 Required Mode

User & Supervisor

10.6.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.6.9 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.

10.6.10 Also See

Minimum, Maximum Value Selection and History Update, Vector Maximum, Vector Minimum (under “Vector Operations”, for Viterbi decode algorithms)

10.6.11 Special Applications

None

10.7 Minimum

10.7.1 General Form

dest_reg = MIN (src_reg_0, src_reg_1)

10.7.2 Syntax

```
Dreg = MIN ( Dreg , Dreg ) ;                     // 32-bit operands (b)
```

10.7.3 Syntax Terminology

Dreg: R0, ..., R7

10.7.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.7.5 Functional 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 operation subtracts src_reg_1 from src_reg_0 and selects the output based on the signs of the input values and the arithmetic flags.

The Minimum instruction does not implicitly modify input values. The dest_reg can be the same D-register as one of the source registers. Doing so explicitly modifies the source register.

10.7.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- V is cleared.

All other flags are unaffected.

10.7.7 Required Mode

User & Supervisor

10.7.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.7.9 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.
```

10.7.10 Also See

Maximum, Vector Maximum, Vector Minimum

10.7.11 Special Applications

None

10.8 Modify – Decrement

10.8.1 General Form

dest_reg -= src_reg

10.8.2 Syntax

40-BIT ACCUMULATORS

```
A0 -= A1 ;              /* dest_reg_new = dest_reg_old - src_reg, saturate the
                        result at 40 bits (b) */

A0 -= A1 (W32) ;        /* dest_reg_new = dest_reg_old - src_reg, decrement and
                        saturate the result at 32 bits, sign extended (b) */
```

32-BIT REGISTERS

```
Preg -= Preg ;          // dest_reg_new = dest_reg_old - src_reg (a)

Ireg -= Mreg ;          // dest_reg_new = dest_reg_old - src_reg (a)
```

10.8.3 Syntax Terminology

Preg: P0, ..., P5, SP, FP

Ireg: I0, ..., I3

Mreg: M0, ..., M3

10.8.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

10.8.5 Functional Description

The Modify-Decrement instruction decrements a register by a user-defined quantity.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

10.8.6 Flags Affected

The Accumulator versions of this instruction affect the flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is set if the operation generates a carry; cleared if no carry.

- AV0 is set if result overflows; cleared if no overflow.
- AV0S is set if AV0 is set; unaffected otherwise.

All other flags are unaffected.

The P-register and I-register versions do not affect any flags.

10.8.7 Required Mode

User & Supervisor

10.8.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

10.8.9 Example

```
a0 -= a1 ;
a0 -= a1 (w32) ;
p3 -= p0 ;
i1 -= m2 ;
```

10.8.10 Also See

Modify – Increment, Subtract, Shift with Add

10.8.11 Special Applications

Typically, use the Index Register and Pointer Register versions of the Modify – Decrement instruction to decrement indirect address pointers for load or store operations.

10.9 Modify – Increment

10.9.1 General Form

dest_reg += src_reg

dest_reg = (src_reg_0 += src_reg_1)

10.9.2 Syntax

40-BIT ACCUMULATORS

A0 += A1 ;	/* dest_reg_new = dest_reg_old + src_reg, saturate the result at 40 bits (b) */
A0 += A1 (W32) ;	/* dest_reg_new = dest_reg_old + src_reg, signed saturate the result at 32 bits, sign extended (b) */

32-BIT REGISTERS

Preg += Preg (BREV) ;	/* dest_reg_new = dest_reg_old + src_reg, bit reversed carry, only (a) */
Ireg += Mreg (opt_brev) ;	/* dest_reg_new = dest_reg_old + src_reg, optional bit reverse (a) */
Dreg = (A0 += A1) ;	/* increment 40-bit A0 by A1 with saturation at 40 bits, then extract the result into a 32-bit register with saturation at 32 bits (b) */

16-BIT HALF-WORD DATA REGISTERS

Dreg_lo_hi = (A0 += A1) ;	/* Increment 40-bit A0 by A1 with saturation at 40 bits, then extract the result into a half register. The extraction step involves first rounding the 40-bit result at bit 16 (according to the RND_MOD bit in the ASTAT register), then saturating at 32 bits and moving bits 31:16 into the half register. (b) */

10.9.3 Syntax Terminology

Dreg: R0, ..., R7

Preg: P0, ..., P5, SP, FP

Ireg: I0, ..., I3

Mreg: M0, ..., M3

opt_brev: optional bit reverse syntax; replace with (brev)

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

10.9.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

10.9.5 Functional Description

The Modify – Increment instruction increments a register by a user-defined quantity. In some versions, the instruction copies the result into a third register.

The “16-bit Half-Word Data Register” version increments the 40-bit A0 by A1 with saturation at 40 bits, then extracts the result into a half register. The extraction step involves first rounding the 40-bit result at bit 16 (according to the RND_MOD bit in the ASTAT register), then saturating at 32 bits and moving bits 31:16 into the half register.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

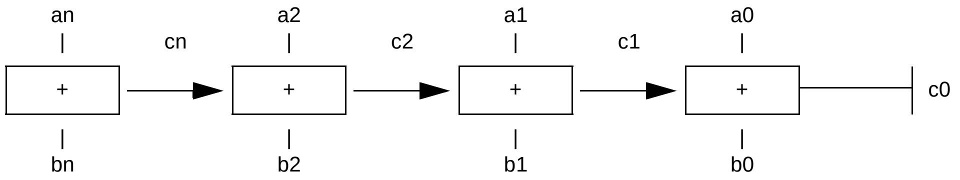
See Section 1.5.6, “Rounding and Truncating,” on page 1-7 for a description of rounding behavior.

10.9.6 Option

(BREV) – bit reverse carry adder. When specified, the carry bit is propagated from left to right, as shown in Figure 10-1, instead of right to left.

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 does not support circular buffering in any case.

Figure 10-1. Bit Addition Flow for the Bit Reverse (BREV) Case



10.9.7 Flags Affected

The Data Register and Accumulator versions of the Modify-Increment instruction affects flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is set if the operation generates a carry; cleared if no carry.
- V is set if result saturates and the dest_reg is a Dreg; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.
- AV0 is set if result saturates and the dest_reg is A0; cleared if no saturation.
- AV0S is set if AV0 is set; unaffected otherwise.

All other flags are unaffected.

The Pointer Register, Index Register, and Modify Register versions of the instruction do not affect the flags.

10.9.8 Required Mode

User & Supervisor

10.9.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

10.9.10 Example

```
a0 += a1 ;
a0 += a1 (w32) ;
p3 += p0 (brev) ;
i1 += m1 ;
i0 += m0 (brev)  ;                              // optional carry bit reverse mode
r5 = (a0 += a1) ;
r2.l = (a0 += a1) ;
r5.h = (a0 += a1) ;
```

10.9.11 Also See

Modify – Decrement, Add, Shift with Add

10.9.12 Special Applications

Typically, use the Index Register and Pointer Register versions of the Modify – Increment instruction to increment indirect address pointers for load or store operations.

10.10 Multiply

10.10.1 General Form

dest_reg = src_reg_0 * src_reg_1 (opt_mode)

10.10.2 Syntax

MULTIPLY-AND-ACCUMULATE UNIT 0 (MAC0)

```
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi  (opt_mode_1) ;    /* 16-bit result into the destination lower
                                                       half-word register (b) */
Dreg = Dreg_lo_hi * Dreg_lo_hi  (opt_mode_2) ;        // 32-bit result (b)
```

MULTIPLY-AND-ACCUMULATE UNIT 1 (MAC1)

```
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi  (opt_mode_1) ;    /* 16-bit result into the destination upper
                                                        half-word register (b) */
Dreg = Dreg_lo_hi * Dreg_lo_hi  (opt_mode_2) ;        // 32-bit result (b)
```

10.10.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo: R0.L, ..., R7.L

Dreg_hi: R0.H, ..., R7.H

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

opt_mode_1: Optionally (FU), (IS), (IU), (T), (TFU), (S2RND), (ISS2) or (IH). Optionally, (M) can be used with MAC1 versions either alone or with any of these other options. When used together, the option flags must be enclosed in one set of parenthesis and separated by a comma. Example: (M, IS).

opt_mode_2: Optionally (FU), (IS), (S2RND), or (ISS2). Optionally, (M) can be used with MAC1 versions either alone or with any of these other options. When used together, the option flags must be enclosed in one set of parenthesis and separated by a comma. Example: (M, IS).

10.10.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.10.5 Functional Description

The Multiply instruction multiplies the two 16-bit operands and stores the result directly into the destination register with saturation.

The instruction is like the Multiply-Accumulate instructions, except that Multiply does not affect the Accumulators.

Operations performed by the Multiply-and-Accumulate Unit 0 (MAC0) portion of the architecture load their results into the lower half of the destination data register. Operations performed by MAC1 load their results into the upper half of the destination data register.

In 32-bit result syntax, the MAC performing the operation will be determined by the destination Dreg. Even-numbered Dregs (R6, R4, R2, R0) invoke MAC0. Odd-numbered Dregs (R7, R5, R3, R1) invoke MAC1. Therefore, 32-bit result operations using the (M) option can only be performed on odd-numbered Dreg destinations.

In 16-bit result syntax, the MAC performing the operation will be determined by the destination Dreg half. Low-half Dregs (R0.L, ..., R7.L) invoke MAC0. High-half Dregs (R0.H, ..., R7.H) invoke MAC1. Therefore, 16-bit result operations using the (M) option can only be performed on high-half Dreg 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.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

See Section 1.5.6, “Rounding and Truncating,” on page 1-7 for a description of rounding behavior.

The versions of this instruction that produce 32-bit results do not perform rounding and are not affected by the RND_MOD bit in the ASTAT register.

10.10.6 Options

The Multiply instruction supports the following options. Saturation is supported for every option.

Table 10-1. Multiply Options (Sheet 1 of 2)

Option	Description
default	both operands of both MACs are treated as signed fractions with left-shift correction to normalize the fraction.
(FU)	unsigned fraction operands. No shift correction.
(IS)	signed integer operands. No shift correction.
(IU)	unsigned integer operands. No shift correction. Available only for the 16-bit destination versions of this instruction.
(T)	signed fraction operands. Truncate the result to 16 bits when copying to the destination half register. Available only for the 16-bit destination versions of this instruction.
(TFU)	unsigned fraction operands. Truncate the result to 16 bits when copying to the destination half register. Available only for the 16-bit destination versions of this instruction.
(S2RND)	signed fraction operands with left-shift correction to normalize the fraction. Scale the result (multiply x2 by performing a one-place shift left) when copying to the destination half register. If scaling produces a signed value larger than 16 bits, the number is saturated to its maximum positive or negative value.

Table 10-1. Multiply Options (Sheet 2 of 2)

Option	Description
(ISS2)	signed integer operands. Scale the result (multiply x2 by performing a one-place shift left) when copying to the destination half register. If scaling produces a signed value larger than 16 bits, the number is saturated to its maximum positive or negative value.
(IH)	integer multiplication with high half-word extraction. The result is saturated at 32 bits and bits 31:16 of that value are copied into the destination half register. Available only for the 16-bit destination versions of this instruction.
(M)	mixed multiply mode. MAC1 multiplies a signed fraction by an unsigned fraction operand with no left-shift correction. Src_reg_0 is signed and src_reg_1 is unsigned. MAC0 performs an unmixed multiply on signed fractions by default or another format as specified. The (M) option can be used alone or in conjunction with one other format option, but only with MAC1 versions of this instruction. When used together, the option flags must be enclosed in one set of parenthesis and separated by a comma. Example: (M, IS).

To truncate the result, the operation eliminates the least significant bits that do not fit into the destination register.

In fractional mode, the product of the smallest representable fraction times itself (i.e., 0x8000 times 0x8000) is saturated to the maximum representable positive fraction (0x7FFF).

10.10.7 Flags Affected

This instruction affects flags as follows:

- V is set if result saturates; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

10.10.8 Required Mode

User & Supervisor

10.10.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.10.10 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. */
```

10.10.11 Also See

Multiply and Multiply-Accumulate to Accumulator, Multiply and Multiply-Accumulate to Half-Register, Multiply and Multiply-Accumulate to Data Register, Multiply (Modulo 2^{32}), Vector Multiply, Vector Multiply and Multiply-Accumulate

10.10.12 Special Applications

10.11 Multiply and Multiply-Accumulate to Accumulator

10.11.1 General Form

```
accumulator = src_reg_0 * src_reg_1  (opt_mode)
accumulator += src_reg_0 * src_reg_1  (opt_mode)
accumulator –= src_reg_0 * src_reg_1  (opt_mode)
```

10.11.2 Syntax

MULTIPLY-AND-ACCUMULATE UNIT 0 (MAC0) OPERATIONS

```
A0 = Dreg_lo_hi * Dreg_lo_hi   (opt_mode) ;            // multiply and store (b)
A0 += Dreg_lo_hi * Dreg_lo_hi   (opt_mode) ;           // multiply and add (b)
A0 –= Dreg_lo_hi * Dreg_lo_hi   (opt_mode) ;           // multiply and subtract (b)
```

MULTIPLY-AND-ACCUMULATE UNIT 1 (MAC1) OPERATIONS

```
A1 = Dreg_lo_hi * Dreg_lo_hi   (opt_mode) ;            // multiply and store (b)
A1 += Dreg_lo_hi * Dreg_lo_hi   (opt_mode) ;           // multiply and add (b)
A1 –= Dreg_lo_hi * Dreg_lo_hi   (opt_mode) ;           // multiply and subtract (b)
```

10.11.3 Syntax Terminology

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

opt_mode: Optionally (FU), (IS), or (W32). Optionally, (M) can be used either alone or with (W32). If multiple options are specified together for a MAC, the options must be separated by commas and enclosed within a single set of parenthesis. Example: (M, W32)

10.11.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.11.5 Functional 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.

10.11.6 Options

The Multiply and Multiply-Accumulate to Accumulator instruction supports the following options. Saturation is supported for every option.

Table 10-2. Options for Multiply and Multiply-Accumulate to Accumulator

Option	Description
(FU)	unsigned fraction or unsigned integer operands. No shift correction.
(IS)	signed integer operands. No shift correction.
(W32)	signed fraction operands, sign extended; saturates both Accumulators at 32 bits. Left-shift correction of the product performed, as required. Used for legacy GSM speech vocoder algorithms written for 32-bit Accumulators. Although the A0.X and A1.X extension bits are not part of computations in this mode, the extension bits are affected through sign extension.
(M)	mixed multiply mode on MAC1. MAC1 multiplies a signed fraction by an unsigned fraction operand with no left-shift correction. Src_reg_0 is signed and src_reg_1 is unsigned. MAC0 performs an unmixed multiply on signed fractions by default or another format as specified. The (M) option can be used alone or in conjunction with the (W32) option for MAC1; no other MAC1 options are valid with (M). When used together, the option flags must be enclosed in one set of parenthesis and separated by a comma. Example: (M, W32).

When the (M) and (W32) options are used together, both MACs saturate their Accumulator products at 32 bits. MAC1 multiplies signed fractions by unsigned fractions and MAC0 multiplies signed fractions.

When used together, the order of the options in the syntax makes no difference.

In fractional mode, the product of the most negative representable fraction times itself (i.e., 0x8000 times 0x8000) is saturated to the maximum representable positive fraction (0x7FFF) before accumulation.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

10.11.7 Flags Affected

This instruction affects flags as follows:

- AV0 is set if result in Accumulator A0 (MAC0 operation) saturates; cleared if A0 result does not saturate.
- AV0S is set if AV0 is set; unaffected otherwise.
- AV1 is set if result in Accumulator A1 (MAC1 operation) saturates; cleared if A1 result does not saturate.
- AV1S is set if AV1 is set; unaffected otherwise.

All other flags are unaffected.

10.11.8 Required Mode

User & Supervisor

10.11.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.11.10 Example

```
a0=r3.h*r2.h ;                        /* MAC0, only. Both operands are signed
                                      fractions. Load the product into A0. */

a1+=r6.h*r4.l (fu) ;                  /* MAC1, only. Both operands are unsigned
                                      fractions. Accumulate into A1 */
```

10.11.11 Also See

Multiply, Multiply and Multiply-Accumulate to Half-Register, Multiply and Multiply-Accumulate to Data Register, Multiply (Modulo 2^{32}), Vector Multiply, Vector Multiply and Multiply-Accumulate

10.11.12 Special Applications

DSP filter applications use the Multiply and Multiply-Accumulate to Accumulator instruction often to calculate the dot product between two signal vectors.

10.12 Multiply and Multiply-Accumulate to Half-Register

10.12.1 General Form

```
dest_reg_half = (accumulator = src_reg_0 * src_reg_1)  (opt_mode)
dest_reg_half = (accumulator += src_reg_0 * src_reg_1)  (opt_mode)
dest_reg_half = (accumulator –= src_reg_0 * src_reg_1)  (opt_mode)
```

10.12.2 Syntax

MULTIPLY-AND-ACCUMULATE UNIT 0 (MAC0)

```
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi)  (opt_mode) ;    // multiply and store (b)
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi)  (opt_mode) ;   // multiply and add (b)
Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi)  (opt_mode) ;   // multiply and subtract (b)
```

MULTIPLY-AND-ACCUMULATE UNIT 1 (MAC1)

```
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi)  (opt_mode) ;    // multiply and store (b)
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi)  (opt_mode) ;   // multiply and add (b)
Dreg_hi = (A1 –= Dreg_lo_hi * Dreg_lo_hi)  (opt_mode) ;   // multiply and subtract (b)
```

10.12.3 Syntax Terminology

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

Dreg_lo: R0.L, ..., R7.L

Dreg_hi: R0.H, ..., R7.H

opt_mode: Optionally (FU), (IS), (IU), (T), (TFU), (S2RND), (ISS2) or (IH). Optionally, (M) can be used with MAC1 versions either alone or with any of these other options. If multiple options are specified together for a MAC, the options must be separated by commas and enclosed within a single set of parenthesis. Example: (M, TFU)

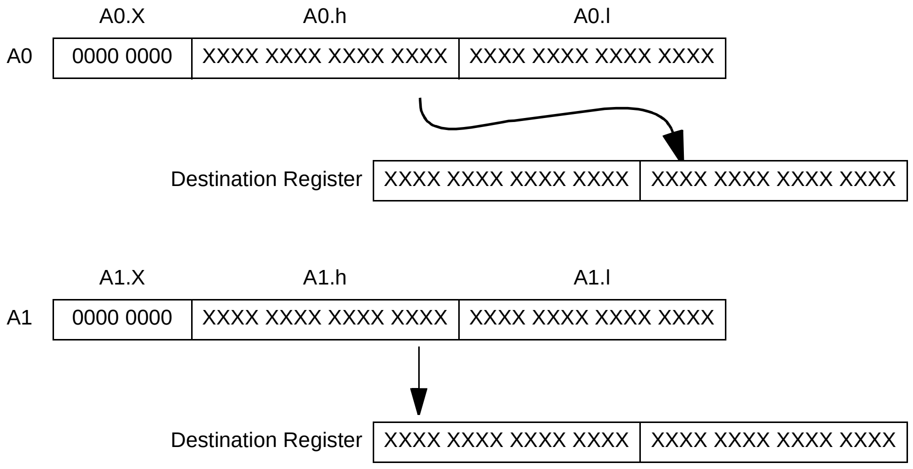
10.12.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

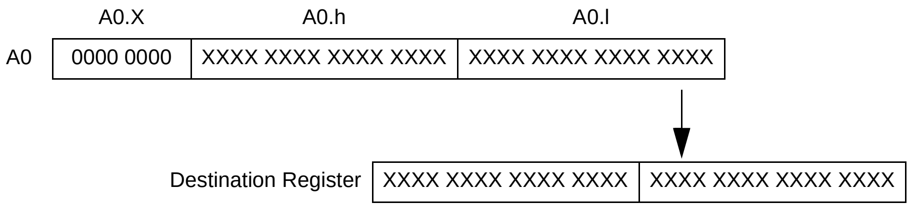
10.12.5 Functional Description

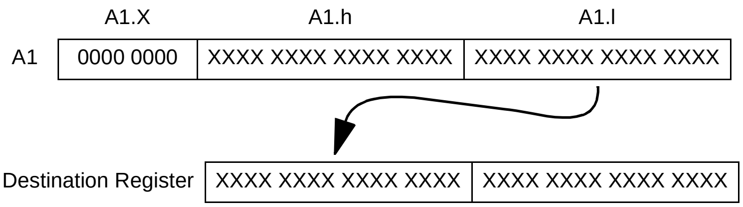
The Multiply and Multiply-Accumulate to Half-Register instruction multiplies two 16-bit half-word operands. The instruction stores, adds or subtracts the product into a designated Accumulator. It then copies 16 bits (saturated at 16 bits) of the Accumulator into a data half-register.

The fraction versions of this instruction (the default and “(FU)” options) transfer the Accumulator result to the destination register according to the diagrams below.


A0.X
A0.h
A0.l
A0
0000 0000
XXXX XXXX XXXX XXXX
XXXX XXXX XXXX XXXX
Destination Register
XXXX XXXX XXXX XXXX
XXXX XXXX XXXX XXXX
A1.X
A1.h
A1.l
A1
0000 0000
XXXX XXXX XXXX XXXX
XXXX XXXX XXXX XXXX
Destination Register
XXXX XXXX XXXX XXXX
XXXX XXXX XXXX XXXX

The integer versions of this instruction (the “(IS)” and “(IU)” options) transfer the Accumulator result to the destination register according to the diagrams, shown below:





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.

See Section 1.5.6, “Rounding and Truncating,” on page 1-7 for a description of rounding behavior.

10.12.6 Options

The Multiply and Multiply-Accumulate to Half-Register instruction supports operand and Accumulator copy options.

The options are:

Table 10-3. Operand and Accumulator Copy Options of Multiply and Multiply-Accumulate to Half-Register (Sheet 1 of 2)

Option	Operand Treatment	Accumulator Copy Formatting
Default	Both operands of both MACs are treated as signed fractions with left-shift correction to normalize the fraction.	High half-word extraction from Accumulator with 16-bit saturation and rounding. The rounding mode is dictated by the RND_MOD bit in the ASTAT register.
(FU)	Unsigned fraction operands. No shift correction.	Same as Default.
(IS)	Signed integer operands. No shift correction.	Low half-word extraction from Accumulator with 16-bit saturation.
(IU)	Unsigned integer operands. No shift correction	Low half-word extraction from Accumulator with 16-bit saturation.

Table 10-3. Operand and Accumulator Copy Options of Multiply and Multiply-Accumulate to Half-Register (Sheet 2 of 2)

Option	Operand Treatment	Accumulator Copy Formatting
(T)	Same as Default.	High half-word extraction with saturation from Accumulator. Truncates low half-word.
(TFU)	Unsigned fraction operands. No shift correction.	High half-word extraction from Accumulator. Truncate low half-word.
(S2RND)	Signed fraction operands with left-shift correction to normalize the fraction.	High half-word extraction with scaling, rounding and 16-bit saturation. The rounding mode is dictated by the RND_MOD bit in the ASTAT register. Scales the Accumulator contents (multiplies x2 by a one-place shift left) and rounds the upper 16 bits before truncating the lower 16 bits.
(ISS2)	Signed integer operands. No shift correction.	Low half-word extraction with scaling and 16-bit saturation. Scales the Accumulator contents (multiplies x2 by a one-place shift left) before copying the lower 16-bits.
(IH)	Signed integer operands. No shift correction.	High half-word extraction with 32-bit saturation, then rounding on upper 16-bits. The rounding mode is dictated by the RND_MOD bit in the ASTAT register.
(M)	Signed by unsigned fraction multiplication. No shift correction. Src_reg_0 is signed and src_reg_1 is unsigned. Only applies to MAC1 versions of this instruction. MAC0 performs an unmixed multiply as directed by option flags (or the default condition if flags are not specified). This option flag can be used alone or in conjunction with one other format option for MAC1. If multiple options are specified together for a MAC, the options must be separated by commas and enclosed within a single set of parenthesis. Example: (M, TFU)	Same as Default.

To truncate the result, the operation eliminates the least significant bits that do not fit into the destination register.

When necessary, saturation is performed after the rounding.

In fractional mode, the product of the most negative representable fraction times itself (i.e., 0x8000 times 0x8000) is saturated to the maximum representable positive fraction (0x7FFF) before accumulation.

If you want to keep the unaltered contents of the Accumulator, use a simple Move instruction to copy A.x or A.w to or from a register.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

10.12.7 Flags Affected

This instruction affects flags as follows:

- V is set if the result extracted to the Dreg saturates; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.
- AV0 is set if result in Accumulator A0 (MAC0 operation) saturates; cleared if A0 result does not saturate.
- AV0S is set if AV0 is set; unaffected otherwise.
- AV1 is set if result in Accumulator A1 (MAC1 operation) saturates; cleared if A1 result does not saturate.
- AV1S is set if AV1 is set; unaffected otherwise.

All other flags are unaffected.

10.12.8 Required Mode

User & Supervisor

10.12.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.12.10 Example

```
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 */
```

10.12.11 Also See

Multiply and Multiply-Accumulate to Accumulator, Multiply and Multiply-Accumulate to Data Register, Multiply (Modulo 2^{32}), Vector Multiply, Vector Multiply and Multiply-Accumulate

10.12.12 Special Applications

DSP filter applications use the Multiply-Accumulate Half-Register instruction often to calculate the dot product between two signal vectors.

10.13 Multiply and Multiply-Accumulate to Data Register

10.13.1 General Form

dest_reg = (accumulator = src_reg_0 * src_reg_1) (opt_mode)

dest_reg = (accumulator += src_reg_0 * src_reg_1) (opt_mode)

dest_reg = (accumulator –= src_reg_0 * src_reg_1) (opt_mode)

10.13.2 Syntax

MULTIPLY-AND-ACCUMULATE UNIT 0 (MAC0)

Dreg_even = (A0 = Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;// multiply and store (b)

Dreg_even = (A0 += Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;// multiply and add (b)

Dreg_even = (A0 –= Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;// multiply and subtract (b)

MULTIPLY-AND-ACCUMULATE UNIT 1 (MAC1)

Dreg_odd = (A1 = Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ; // multiply and store (b)

Dreg_odd = (A1 += Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;// multiply and add (b)

Dreg_odd = (A1 –= Dreg_lo_hi * Dreg_lo_hi) (opt_mode) ;// multiply and subtract (b)

10.13.3 Syntax Terminology

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

Dreg_even: R0, R2, R4, R6

Dreg_odd: R1, R3, R5, R7

opt_mode: Optionally (FU), (IS), (S2RND), or (ISS2). Optionally, (M) can be used with MAC1 versions either alone or with any of these other options. If multiple options are specified together for a MAC, the options must be separated by commas and enclosed within a single set of parenthesis. Example: (M, IS)

10.13.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.13.5 Functional Description

This instruction multiplies two 16-bit half-word operands. The instruction stores, adds or subtracts the product into a designated Accumulator. It then copies 32 bits of the Accumulator into a data register. The 32 bits are saturated at 32 bits.

The Multiply-and-Accumulate Unit 0 (MAC0) portion of the architecture performs operations that involve Accumulator A0; it loads the results into an even-numbered data register. MAC1 performs A1 operations and loads the results into an odd-numbered data register.

Combinations of these instructions can be combined into a single instruction. See Section 14.10, “Vector Multiply and Multiply-Accumulate,” on page 14-28.

10.13.6 Options

The Multiply and Multiply-Accumulate to Data Register instruction supports operand and Accumulator copy options.

The options are:

Table 10-4. Operand and Accumulator Copy Options of Multiply and Multiply-Accumulate to Data Register

Option	Operand Treatment	Accumulator Copy Formatting
Default	Both operands of both MACs are treated as signed fractions with left-shift correction to normalize the fraction.	32-bit extraction from Accumulator with 32-bit saturation.
(FU)	Unsigned fraction operands. No shift correction.	Same as Default.
(IS)	Signed integer operands. No shift correction.	Same as Default.
(S2RND)	Signed fraction operands with left-shift correction to normalize the fraction.	32-bit extraction with scaling and 32-bit saturation. Scales the Accumulator contents (multiplies x2 by a one-place shift left).
(ISS2)	Signed integer operands. No shift correction.	32-bit extraction with scaling and 32-bit saturation. Scales the Accumulator contents (multiplies x2 by a one-place shift left).
(M)	Signed by unsigned fraction multiplication. No shift correction. Src_reg_0 is signed and src_reg_1 is unsigned. Only applies to MAC1 versions of this instruction. MAC0 performs an unmixed multiply as directed by option flags (or the default condition if flags are not specified). This option flag can be used alone or in conjunction with one other format option for MAC1. If multiple options are specified together for a MAC, the options must be separated by commas and enclosed within a single set of parenthesis. Example: (M, IS)	Same as Default.

The syntax supports only biased rounding. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

See Section 1.5.6, “Rounding and Truncating,” on page 1-7 for a description of rounding behavior.

In fractional mode, the product of the most negative representable fraction times itself (i.e., 0x8000 times 0x8000) is saturated to the maximum representable positive fraction (0x7FFF) before accumulation.

If you want to keep the unaltered contents of the Accumulator, use a simple Move instruction to copy A.x or A.w to or from a register.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

10.13.7 Flags Affected

This instruction affects flags as follows:

- V is set if the result extracted to the Dreg saturates; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.
- AV0 is set if result in Accumulator A0 (MAC0 operation) saturates; cleared if A0 result does not saturate.
- AV0S is set if AV0 is set; unaffected otherwise.
- AV1 is set if result in Accumulator A1 (MAC1 operation) saturates; cleared if A1 result does not saturate.
- AV1S is set if AV1 is set; unaffected otherwise.

All other flags are unaffected.

10.13.8 Required Mode

User & Supervisor

10.13.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.13.10 Example

```
r4=(a0=r3.h*r2.h) ;                  /* MAC0, only. Both operands are signed
                                     fractions. Load the product into A0, then into
                                      r4. */
r3=(a1+=r6.h*r4.l) (fu) ;            /* MAC1, only. Both operands are unsigned
                                     fractions. Add the product into A1, then into
                                      r3. */
```

10.13.11 Also See

Move Register, Move Register Half, Multiply and Multiply-Accumulate to Accumulator, Multiply and Multiply-Accumulate to Half-Register, Multiply (Modulo 2^{32}), Vector Multiply, Vector Multiply and Multiply-Accumulate

10.13.12 Special Applications

DSP filter applications often use the Multiply and Multiply-Accumulate to Data Register instruction or the vector version (“Vector Multiply and Multiply-Accumulate” on page 14-28) to calculate the dot product between two signal vectors.

10.14 Multiply (Modulo 2^{32})

10.14.1 General Form

dest_reg *= multiple_register

10.14.2 Syntax

Dreg *= Dreg ;	// 32 x 32 integer multiply (a)

10.14.3 Syntax Terminology

Dreg: R0, ..., R7

10.14.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

10.14.5 Functional Description

The Multiply (Modulo 2^{32}) instruction multiplies two 32-bit data registers -- dest_reg and multiple_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 instruction is modulo 2^{32}, the result always fits in a 32-bit dest_reg, and overflows are not possible; the overflow flag 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 (Modulo 2^{32}) instruction does not implicitly modify the number in multiple_register.

This instruction might be used to implement the congruence method of random number generation according to...

x[n+1] = (a*x[n]) mod 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.

10.14.6 Flags Affected

None

10.14.7 Required Mode

User & Supervisor

10.14.8 Parallel Issue

This instruction cannot be issued in parallel with any other instructions.

10.14.9 Example

```
r3 *= r0 ;
```

10.14.10 Also See

Divide Primitive, Arithmetic Shift, Shift with Add, Add with Shift

In the “Vector Operations” chapter, see Vector Multiply and Multiply-Accumulate, Vector Multiply

10.14.11 Special Applications

None

10.15 Negate (Two’s Complement)

10.15.1 General Form

dest_reg = – src_reg

dest_accumulator = – src_accumulator

10.15.2 Syntax

```
Dreg = – Dreg ;                      // (a)
Dreg = – Dreg (sat_flag) ;           // (b)
A0 = – A0 ;                          // (b)
A0 = – A1 ;                          // (b)
A1 = – A0 ;                          // (b)
A1 = – A1 ;                          // (b)
A1 = – A1, A0 = – A0 ;              /* negate both Accumulators simultaneously in
                                     one 32-bit length instruction (b) */
```

10.15.3 Syntax Terminology

Dreg: R0, ..., R7

sat_flag: non-optional saturation flag, (S) or (NS)

10.15.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

10.15.5 Functional 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.

The Dreg version without the sat_flag does not saturate. The only case where the non-saturating Negate would overflow is when the input value is 0x8000 0000. The saturating version returns 0x7FFF FFFF; the non-saturating version returns 0x8000 0000.

In the syntax, where sat_flag appears, substitute one of the following values:

- (S) – saturate the result
- (NS) – no saturation

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

10.15.6 Flags Affected

This instruction affects the flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- V is set if result overflows or saturates and the dest_reg is a Dreg; cleared if no overflow or saturation.
- VS is set if V is set; unaffected otherwise.
- AV0 is set if result saturates and the dest_reg is A0; cleared if no saturation.
- AV0S is set if AV0 is set; unaffected otherwise.
- AV1 is set if result saturates and the dest_reg is A1; cleared if no saturation.
- AV1S is set if AV1 is set; unaffected otherwise.

All other flags are unaffected.

10.15.7 Required Mode

User & Supervisor

10.15.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

10.15.9 Example

```
r5 =-r0 ;
a0 =-a0 ;
a0 =-a1 ;
a1 =-a0 ;
a1 =-a1 ;
a1 =-a1, a0=-a0 ;
```

10.15.10 Also See

Vector Negate (Two’s Complement) (in “Vector Operations” chapter)

10.15.11 Special Applications

None

10.16 Round Half-Word

10.16.1 General Form

dest_reg = src_reg (RND)

10.16.2 Syntax

Dreg_lo_hi =Dreg (RND) ; // round and saturate the source to 16 bits. (b)

10.16.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

10.16.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.16.5 Functional Description

The Round Half-Word instruction rounds a 32-bit, normalized-fraction number into a 16-bit, normalized-fraction 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.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

See Section 1.5.6, “Rounding and Truncating,” on page 1-7 for a description of rounding behavior.

10.16.6 Flags Affected

The following flags are affected by this instruction:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- V is set if result saturates; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

10.16.7 Required Mode

User & Supervisor

10.16.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.16.9 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
```

10.16.10 Also See

Round – 12 Bit, Round – 20 Bit, Add

10.16.11 Special Applications

None

10.17 Round – 12 Bit

10.17.1 General Form

dest_reg = src_reg_0 + src_reg_1 (RND12)

dest_reg = src_reg_0 - src_reg_1 (RND12)

10.17.2 Syntax

```
Dreg_lo_hi = Dreg + Dreg (RND12) ;          // (b)
Dreg_lo_hi = Dreg - Dreg (RND12) ;          // (b)
```

10.17.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

10.17.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.17.5 Functional Description

The Round – 12 Bit instruction adds or subtracts two 32-bit values, then rounds the sum on bit position 12. The instruction saves 16 bits of the number by extracting bits 27:12 and saturates the result.

The instruction supports only biased rounding, which adds a half LSB (in this case bit 11) before truncating bits 11:0. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

See Section 1.5.6, “Rounding and Truncating,” on page 1-7 for a description of rounding behavior.

10.17.6 Flags Affected

The following flags are affected by this instruction:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- V is set if result saturates; cleared if no saturation.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

10.17.7 Required Mode

User & Supervisor

10.17.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.17.9 Example

```
r1.l = r6+r7(rnd12) ;
r1.l = r6-r7(rnd12) ;
r1.h = r6+r7(rnd12) ;
r1.h = r6-r7(rnd12) ;
```

10.17.10 Also See

Round – Half-Word, Round – 20 Bit, Add

10.17.11 Special Applications

Typically, use the Round – 12 Bit instruction to provide an IEEE 1180–compliant 2D 8x8 inverse discrete cosine transform.

10.18 Round – 20 Bit

10.18.1 General Form

dest_reg = src_reg_0 + src_reg_1 (RND20)

dest_reg = src_reg_0 - src_reg_1 (RND20)

10.18.2 Syntax

```
Dreg_lo_hi = Dreg + Dreg (RND20) ;          // (b)
Dreg_lo_hi = Dreg - Dreg (RND20) ;          // (b)
```

10.18.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

10.18.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.18.5 Functional Description

The Round – 20 Bit instruction effectively adds or subtracts two 32-bit values, rounds the result at bit 20, then extracts bits 31:20 into a 16-bit destination register. Round – 20 Bit supports only biased rounding, which adds a half LSB (in this case, bit 19) before truncating bits 19:0.

In practice, this instruction right-shifts each input term four bits to prevent overflow of the result, adds or subtracts the two terms, then rounds the result on bit position 16. The instruction saves the upper 16 bits of the number.

The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

See Section 1.5.6, “Rounding and Truncating,” on page 1-7 for a description of rounding behavior.

10.18.6 Flags Affected

The following flags are affected by this instruction:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- V is cleared.

All other flags are unaffected.

10.18.7 Required Mode

User & Supervisor

10.18.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.18.9 Example

```
r1.l = r6+r7(rnd20) ;
r1.l = r6-r7(rnd20) ;
r1.h = r6+r7(rnd20) ;
r1.h = r6-r7(rnd20) ;
```

10.18.10 Also See

Round – 12 Bit, Round – Half-Word, Add

10.18.11 Special Applications

Typically, use the Round 20 – Bit instruction to provide an IEEE 1180–compliant 2D 8x8 inverse discrete cosine transform.

10.19 Saturate

10.19.1 General Form

dest_reg = src_reg (S)

10.19.2 Syntax

```
A0 = A0 (S) ;                         // (b)
A1 = A1 (S) ;                         // (b)
A1 = A1 (S), A0 = A0 (S) ;            /* signed saturate both Accumulators at
                                      the 32-bit boundary (b) */
```

10.19.3 Syntax Terminology

None

10.19.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.19.5 Functional Description

The Saturate instruction saturates the 40-bit Accumulators at 32 bits. The resulting saturated value is sign extended into the Accumulator extension bits.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

10.19.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if result is zero; cleared if non-zero. In the case of two simultaneous operations, AZ represents the logical “OR” of the two.
- AN is set if result is negative; cleared if non-negative. n the case of two simultaneous operations, AN represents the logical “OR” of the two.
- AV0 is set if result saturates and the dest_reg is A0; cleared if no overflow.
- AV0S is set if AV0 is set; unaffected otherwise.
- AV1 is set if result saturates and the dest_reg is A1; cleared if no overflow.
- AV1S is set if AV1 is set; unaffected otherwise.

All other flags are unaffected.

10.19.7 Required Mode

User & Supervisor

10.19.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.19.9 Example

```
a0 = a0 (s) ;
a1 = a1 (s) ;
a1 = a1 (s), a0 = a0 (s) ;
```

10.19.10 Also See

Subtract (saturate options), Add (saturate options)

10.19.11 Special Applications

None

10.20 Sign Bit

10.20.1 General Form

dest_reg = SIGNBITS sample_register

10.20.2 Syntax

```
Dreg_lo = SIGNBITS Dreg ;              // 32-bit sample (b)
Dreg_lo = SIGNBITS Dreg_lo_hi ;        // 16-bit sample (b)
Dreg_lo = SIGNBITS A0 ;                // 40-bit sample (b)
Dreg_lo = SIGNBITS A1 ;                // 40-bit sample (b)
```

10.20.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo: R0.L, ..., R7.L

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

10.20.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

10.20.5 Functional Description

The Sign Bit instruction returns the number of sign bits in a number, and can be used in conjunction with a shift to normalize numbers. It can operate on 16-bit, 32-bit, or 40-bit input numbers.

- For a 16-bit input, Sign Bit returns the number of leading signbits minus one, which is in the range 0 to 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 signbits minus one, which is in the range 0 to 31. An input of all zeros or all ones returns +31 (all sign bits).
- For a 40-bit Accumulator input, Sign Bit returns the number of leading signbits minus 9, which is in the range -8 to 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 SIGNBITS instruction can be used directly as the argument to 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
40-bit:	SSSS SSSS S.MMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM

In addition, the SIGNBITS result can be subtracted directly to form the new exponent.

The Sign Bit instruction does not implicitly modify the input value. For 32-bit and 16-bit input, the dest_reg and sample_register can be the same D-register. So doing explicitly modifies the sample_register.

10.20.6 Flags Affected

None

10.20.7 Required Mode

User & Supervisor

10.20.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.20.9 Example

```
r2.l = signbits r7 ;
r1.l = signbits r5.l ;
r0.l = signbits r4.h ;
r6.l = signbits a0 ;
r5.l = signbits a1 ;
```

10.20.10 Also See

Exponent Detection

10.20.11 Special Applications

You can use the exponent as shift magnitude for array normalization. You can accomplish normalization by using the ASHIFT instruction directly, without using special normalizing instructions, as required on other architectures.

10.21 Subtract

10.21.1 General Form

dest_reg = src_reg_1 - src_reg_2

10.21.2 Syntax

32-BIT OPERANDS, 32-BIT RESULT

Dreg = Dreg - Dreg ;	/* no saturation support but shorter instruction length (a) */
Dreg = Dreg - Dreg (sat_flag) ;	/* saturation optionally supported, but at the cost of longer instruction length (b) */

16-BIT OPERANDS, 16-BIT RESULT

Dreg_lo_hi = Dreg_lo_hi – Dreg_lo_hi (sat_flag) ; // (b)

10.21.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

sat_flag: non-optional saturation flag, (S) or (NS)

10.21.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

10.21.5 Functional Description

The Subtract instruction subtracts src_reg_2 from src_reg_1 and places the result in a destination register.

There are two ways to specify subtraction on 32-bit data. One that is 16-bit instruction length does not support saturation. The other instruction, which is 32-bit instruction length, optionally supports saturation. The larger DSP instruction can sometimes save execution time because it can be issued in parallel with certain other instructions. See “Parallel Issue”.

The instructions for 16-bit data use half-word data register operands and store the result in a half-word data register.

All the instructions for 16-bit data are 32-bit instruction length.

In the syntax, where sat_flag appears, substitute one of the following values:

- (S) – saturate the result

- (NS) – no saturation

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

The Subtract instruction has no subtraction equivalent of the addition syntax for P-registers.

10.21.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if result is zero; cleared if non-zero.
- AN is set if result is negative; cleared if non-negative.
- AC0 is set if the operation generates a carry; cleared if no carry.
- V is set if result overflows; cleared if no overflow.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

10.21.7 Required Mode

User & Supervisor

10.21.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

The 16-bit versions of this instruction cannot be issued in parallel with other instructions.

10.21.9 Example

```
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

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) ;
```

10.21.10 Also See

Modify – Decrement, Vector Add/Subtract (in “Vector Operations” chapter)

10.21.11 Special Applications

None

10.22 Subtract Immediate

10.22.1 General Form

register -= constant

10.22.2 Syntax

```
Ireg -= 2 ;                    /* decrement Ireg by 2, half-word address
                               pointer increment (a) */

Ireg -= 4 ;                    // word address pointer decrement (a)
```

10.22.3 Syntax Terminology

Ireg: I0, ..., I3

10.22.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

10.22.5 Functional Description

The Subtract Immediate instruction subtracts a constant value from anIndex register without saturation.

Note: To subtract immediate values from D-registers or P-registers, use a negative constant in the Add Immediate instruction.

10.22.6 Flags Affected

None

10.22.7 Required Mode

User & Supervisor

10.22.8 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

10.22.9 Example

```
i0 -= 4 ;
```

```
i2 -= 2 ;
```

10.22.10 Also See

Add Immediate, Subtract

10.22.11 Special Applications

None

EXTERNAL EVENT MANAGEMENT 11

Instruction Summary

This chapter discusses the instructions that manage external events. Users can take advantage of these instructions to enable interrupts, force a specific interrupt or reset to occur, or put the processor in idle state. The Core Synchronize instruction resolves all pending operations and flushes the core store buffer before proceeding to the next instruction. The System Synchronize instruction forces all speculative, transient states in the core and system to complete before processing continues. Other instructions in this chapter force an emulation exception, placing the processor in Emulation mode, test the value of a specific, indirectly addressed byte, or increment the Program Counter without performing useful work.

11.1 Idle

11.1.1 General Form

IDLE

11.1.2 Syntax

```
IDLE ;                              // (a)
```

11.1.3 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

11.1.4 Functional Description

Typically, the Idle instruction is part of a sequence to place the Blackfin in a quiescent state so that the external system can switch between core clock frequencies.

The Idle instruction requests an idle state by setting the idle_req bit in SEQSTAT register. Setting the idle_req bit precedes placing the Blackfin in a quiescent state. If you intend to place the processor in Idle mode, the IDLE instruction must immediately precede a SSYNC instruction.

The first instruction following the SSCYNC is the first instruction to execute when the processor recovers from Idle mode.

The Idle instruction is the only way to set the idle_req bit in SEQSTAT. The architecture does not support explicit writes to SEQSTAT.

11.1.5 Flags Affected

None

11.1.6 Required Mode

The Idle instruction executes only in Supervisor mode. If execution is attempted in User mode, the instruction produces an Illegal Use of Protected Resource exception.

11.1.7 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

11.1.8 Example

```
idle ;
```

11.1.9 Also See

System Synchronize

11.1.10 Special Applications

11.2 Core Synchronize

11.2.1 General Form

CSYNC

11.2.2 Syntax

```
CSYNC ;                         // (a)
```

11.2.3 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

11.2.4 Functional Description

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 MMR writes to prevent imprecise behavior.

11.2.5 Flags Affected

None

11.2.6 Required Mode

User & Supervisor

11.2.7 Parallel Issue

The Core Synchronize instruction cannot be issued in parallel with other instructions.

11.2.8 Example

Consider the following example code sequence.

```
if cc jump away_from_here  ;          // produces speculative branch prediction
csync ;
r0 = [p0]  ;                          // load
```

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.

11.2.9 Also See

System Synchronize

11.2.10 Special Applications

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 MMRs to make sure the data reaches the MMR before the next instruction is fetched.

Typically, the Blackfin 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 that stores never access memory speculatively.

11.3 System Synchronize

11.3.1 General Form

SSYNC

11.3.2 Syntax

```
SSYNC ;                              // (a)
```

11.3.3 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

11.3.4 Functional Description

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 Synch request signal to the external system. The operation requires an acknowledgement Synch_Ack signal by the system before completing the instruction.

If the idle_req bit of the SEQSTAT register is set when SSYNC is executed, the processor enters Idle state and asserts the external Idle signal after receiving the external Synch_Ack signal. After the external Idle signal is asserted, exiting the Idle state requires an external Wakeup signal.

SSYNC should be issued immediately before and after writing to a system MMR. Otherwise, the MMR change can take effect at an indeterminate time while other instructions are executing, resulting in imprecise behavior.

11.3.5 Flags Affected

None

11.3.6 Required Mode

User & Supervisor

11.3.7 Parallel Issue

The SSYNC instruction cannot be issued in parallel with other instructions.

11.3.8 Example

Consider the following example code sequence.

```
if cc jump away_from_here  ;      // produces speculative branch prediction
ssync ;
r0 = [p0]  ;                      // load
```

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.

11.3.9 Also See

Core Synchronize, Idle

11.3.10 Special Applications

Typically, SSYNC prepares the architecture for clock cessation or frequency change. In such cases, the following instruction sequence is typical:

```
:
instruction...
instruction...
CLI r0 ;                          // disable interrupts
idle ;                            // enable Idle state
ssync ;                           /* conclude all speculative states, assert
                                  external Sync signal, await Synch_Ack,
                                  then assert external Idle signal and stall
                                  in the Idle state until the Wakeup signal.
                                  Clock input can be modified during the
                                  stall. */
sti r0 ;                          /* re-enable interrupts when Wakeup
                                  occurs */
instruction...
instruction...
```

11.4 Force Emulation

11.4.1 General Form

EMUEXCPT

11.4.2 Syntax

```
EMUEXCPT ;                          // (a)
```

11.4.3 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

11.4.4 Functional Description

The Force Emulation instruction forces an emulation exception, thus 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 generates an illegal instruction exception.

An emulation exception is the highest priority event in the processor.

11.4.5 Flags Affected

None

11.4.6 Required Mode

User & Supervisor

11.4.7 Parallel Issue

The Force Emulation instruction cannot be issued in parallel with other instructions.

11.4.8 Example

```
emuexcpt ;
```

11.4.9 Also See

Force Interrupt / Reset

11.4.10 Special Applications

11.5 Disable Interrupts

11.5.1 General Form

CLI

11.5.2 Syntax

```
CLI Dreg ;                    // previous state of IMASK moved to Dreg (a)
```

11.5.3 Syntax Terminology

Dreg: R0, ..., R7

11.5.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

11.5.5 Functional Description

The Disable Interrupts instruction globally disables general interrupts by setting IMASK 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.

11.5.6 Flags Affected

None

11.5.7 Required Mode

The Disable Interrupts instruction executes only in Supervisor mode. If execution is attempted in User mode, the instruction produces an Illegal Use of Protected Resource exception.

11.5.8 Parallel Issue

The Disable Interrupts instruction cannot be issued in parallel with other instructions.

11.5.9 Example

```
cli r3 ;
```

11.5.10 Also See

Enable Interrupts

11.5.11 Special Applications

This instruction is often issued immediately before an IDLE instruction.

11.6 Enable Interrupts

11.6.1 General Form

STI

11.6.2 Syntax

```
STI Dreg ;                    // previous state of IMASK restored from Dreg (a)
```

11.6.3 Syntax Terminology

Dreg: R0, ..., R7

11.6.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

11.6.5 Functional Description

The Enable Interrupts instruction globally enables interrupts by restoring the previous state of the interrupt system back into IMASK.

11.6.6 Flags Affected

None

11.6.7 Required Mode

The Enable Interrupts instruction executes only in Supervisor mode. If execution is attempted in User mode, the instruction produces an Illegal Use of Protected Resource exception.

11.6.8 Parallel Issue

The Enable Interrupts instruction cannot be issued in parallel with other instructions.

11.6.9 Example

```
sti r3 ;
```

11.6.10 Also See

Disable Interrupts

11.6.11 Special Applications

This instruction is often located after an IDLE instruction so that it will execute after a wakeup event from the idle state.

11.7 Force Interrupt / Reset

11.7.1 General Form

RAISE

11.7.2 Syntax

```
RAISE uimm4 ;                    // (a)
```

11.7.3 Syntax Terminology

uimm4: 4-bit unsigned field, with the range of 0 through 15

11.7.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

11.7.5 Functional Description

The Force Interrupt / Reset instruction forces a specified interrupt or reset 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. This instruction causes these events to occur given the uimm4 arguments shown:

0. <reserved>
1. RST
2. NMI
3. <reserved>
4. <reserved>
5. IVHW
6. IVTMR
7. ICG7
8. IVG8
9. IVG9
10. IVG10
11. IVG11
12. IVG12
13. IVG13

14. IVG14
15. IVG15

The Force Interrupt / Reset 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.

11.7.6 Flags Affected

None

11.7.7 Required Mode

The Force Interrupt / Reset 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.

11.7.8 Parallel Issue

The Force Interrupt / Reset instruction cannot be issued in parallel with other instructions.

11.7.9 Example

```
raise 1 ;                              // Invoke RST
raise 6 ;                              // Invoke IVTMR timer interrupt
```

11.7.10 Also See

Force Exception (EXCPT), Force Emulation (EMUEXCPT)

11.7.11 Special Applications

11.8 Force Exception

11.8.1 General Form

EXCPT

11.8.2 Syntax

```
EXCPT uimm4 ;                    // (a)
```

11.8.3 Syntax Terminology

uimm4: 4-bit unsigned field, with the range of 0 through 15

11.8.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

11.8.5 Functional Description

The Force Exception instruction forces an exception with code uimm4. 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.

11.8.6 Flags Affected

None

11.8.7 Required Mode

User & Supervisor

11.8.8 Parallel Issue

The Force Exception instruction cannot be issued in parallel with other instructions.

11.8.9 Example

```
excpt 4 ;
```

11.8.10 Also See

11.8.11 Special Applications

11.9 Test and Set Byte (Atomic)

11.9.1 General Form

TESTSET

11.9.2 Syntax

```
TESTSET ( Preg ) ;                    // (a)
```

11.9.3 Syntax Terminology

Preg: P0, ..., P5 (SP and FP are not allowed as the register for this instruction)

11.9.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

11.9.5 Functional Description

The Test and Set Byte (Atomic) instruction loads an indirectly addressed memory byte, tests whether it is zero, then sets the most significant bit of the memory byte without affecting any other bits. If the byte is originally zero, the instruction sets the CC bit. If the byte is originally non-zero the instruction clears the CC bit. The sequence of this memory transaction is *atomic*.

TESTSET accesses the entire logical memory space except the core Memory-Mapped Register (MMR) address region. The system design must ensure atomicity for all memory regions that TESTSET may access. The hardware does not perform atomic access to L1 memory space configured as SRAM. Therefore, semaphores must not reside in on-core memory.

The memory architecture always treats atomic operations as cache-inhibited accesses, even if the CPLB descriptor for the address indicates a cache-enabled access. If a cache hit is detected, the operation flushes and invalidates the line before allowing the TESTSET to proceed.

The software designer is responsible for executing atomic operations in the proper cacheable / non-cacheable memory space. Typically, these operations should execute in non-cacheable, off-core memory. In a chip implementation that requires tight temporal coupling between processors or processes, the design should implement a dedicated, non-cacheable block of memory that meets the data latency requirements of the system.

TESTSET can be interrupted before the load portion of the instruction completes. If interrupted, the TESTSET will be re-executed upon return from the interrupt. After the test, or load, portion of the TESTSET completes, the TESTSET sequence cannot be interrupted. For example, any exceptions associated with the CPLB lookup for both the load and store operations must be completed before the load of the TESTSET completes.

The integrity of the TESTSET atomicity depends on the L2 memory resource-locking mechanism. If the L2 memory does not support atomic locking for the address region you are accessing, your software has no guarantee of correct semaphore behavior. See the processor L2 memory documentation for more on the locking support.

11.9.6 Flags Affected

This instruction affects flags as follows:

- CC is set if addressed value is zero; cleared if non-zero.

All other flags are unaffected.

11.9.7 Required Mode

User and Supervisor

11.9.8 Parallel Issue

The TESTSET instruction cannot be issued in parallel with other instructions.

11.9.9 Example

testset (p1) ;

The TESTSET instruction may be preceded by a CSYNC or SSYNC instruction to ensure that all previous exceptions or interrupts have been processed before the atomic operation begins.

11.9.10 Also See

Core Synchronize, System Synchronize

11.9.11 Special Applications

Typically, use TESTSET as a semaphore sampling method between co-processors or co-processes.

11.10 No Op

11.10.1 General Form

NOP
MNOP

11.10.2 Syntax

```
NOP ;                                   // (a)
MNOP ;                                  // (b)
```

11.10.3 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length. Comment (b) identifies 32-bit instruction length.

11.10.4 Functional 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.

11.10.5 Flags Affected

None

11.10.6 Required Mode

User & Supervisor

11.10.7 Parallel Issue

The 16-bit versions of this instruction can be issued in parallel with specific other instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

11.10.8 Example

```
nop ;

mnop ;
```

11.10.9 Also See

11.10.10 Special Applications

MNOP can be used to issue loads or store instructions in parallel without invoking a 32-bit MAC or ALU operation. Refer to Chapter 15, “Issuing Parallel Instructions,” for more information.

CACHE CONTROL 12

Instruction Summary

This chapter discusses the instructions that control cache. Users can take advantage of these instructions to prefetch or flush the data cache, invalidate data cache lines, or flush the instruction cache.

12.1 Data Cache Prefetch

12.1.1 General Form

PREFETCH

12.1.2 Syntax

```
PREFETCH [ Preg ] ;            // indexed (a)
PREFETCH [ Preg ++ ] ;         // indexed, post increment (a)
```

12.1.3 Syntax Terminology

Preg: P0, ..., P5, SP, FP

12.1.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

12.1.5 Functional Description

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.

Option: The instruction can post-increment the line pointer by the cache-line size.

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.

12.1.6 Flags Affected

None

12.1.7 Required Mode

User & Supervisor

12.1.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

12.1.9 Example

```
prefetch [ p2 ] ;
prefetch [ p0 ++ ] ;
```

12.1.10 Also See

12.1.11 Special Applications

12.2 Data Cache Flush

12.2.1 General Form

FLUSH

12.2.2 Syntax

```
FLUSH [ Preg ] ;            // indexed (a)
FLUSH [ Preg ++ ] ;         // indexed, post increment (a)
```

12.2.3 Syntax Terminology

Preg: P0, ..., P5, SP, FP

12.2.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

12.2.5 Functional Description

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.

Option: The instruction can post-increment the line pointer by the cache-line size.

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.

12.2.6 Flags Affected

None

12.2.7 Required Mode

User & Supervisor

12.2.8 Parallel Issue

The instruction cannot be issued in parallel with other instructions.

12.2.9 Example

```
flush [ p2 ] ;
flush [ p0 ++ ] ;
```

12.2.10 Also See

12.2.11 Special Applications

12.3 Data Cache Line Invalidate

12.3.1 General Form

FLUSHINV

12.3.2 Syntax

```
FLUSHINV [ Preg ] ;          // indexed (a)
FLUSHINV [ Preg ++ ] ;       // indexed, post increment (a)
```

12.3.3 Syntax Terminology

Preg: P0, ..., P5, SP, FP

12.3.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

12.3.5 Functional Description

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.

Option: The instruction can post-increment the line pointer by the cache-line size.

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.

12.3.6 Flags Affected

None

12.3.7 Required Mode

User & Supervisor

12.3.8 Parallel Issue

The Data Cache Line Invalidate instruction cannot be issued in parallel with other instructions.

12.3.9 Example

```
flushinv [ p2 ] ;
flushinv [ p0 ++ ] ;
```

12.3.10 Also See

12.3.11 Special Applications

12.4 Instruction Cache Flush

12.4.1 General Form

IFLUSH

12.4.2 Syntax

```
IFLUSH [ Preg ] ;                // indexed (a)
IFLUSH [ Preg ++ ] ;             // indexed, post increment (a)
```

12.4.3 Syntax Terminology

Preg: P0, ..., P5, SP, FP

12.4.4 Instruction Length

In the syntax, comment (a) identifies 16-bit instruction length.

12.4.5 Functional Description

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.

Option: The instruction can post-increment the line pointer by the cache-line size.

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.

12.4.6 Flags Affected

None

12.4.7 Required Mode

User & Supervisor

12.4.8 Parallel Issue

This instruction cannot be issued in parallel with other instructions.

12.4.9 Example

```
iflush [ p2 ] ;
iflush [ p0 ++ ] ;
```

12.4.10 Also See

12.4.11 Special Applications

VIDEO PIXEL OPERATIONS 13

Instruction Summary

This chapter discusses the instructions that manipulate video pixels. Users can take advantage of these instructions to align bytes, disable exceptions that result from misaligned 32-bit memory accesses, and perform dual and quad 8- and 16-bit add, subtract, and averaging operations.

13.1 Byte Align

13.1.1 General Form

dest_reg = ALIGN8 (src_reg_1, src_reg_0)
dest_reg = ALIGN16 (src_reg_1, src_reg_0)
dest_reg = ALIGN24 (src_reg_1, src_reg_0)

13.1.2 Syntax

```
Dreg = ALIGN8 ( Dreg , Dreg ) ;           // overlay 1 byte (b)
Dreg = ALIGN16 ( Dreg , Dreg )  ;         // overlay 2 bytes (b)
Dreg = ALIGN24 ( Dreg , Dreg )  ;         // overlay 3 bytes (b)
```

13.1.3 Syntax Terminology

Dreg: R0, ..., R7

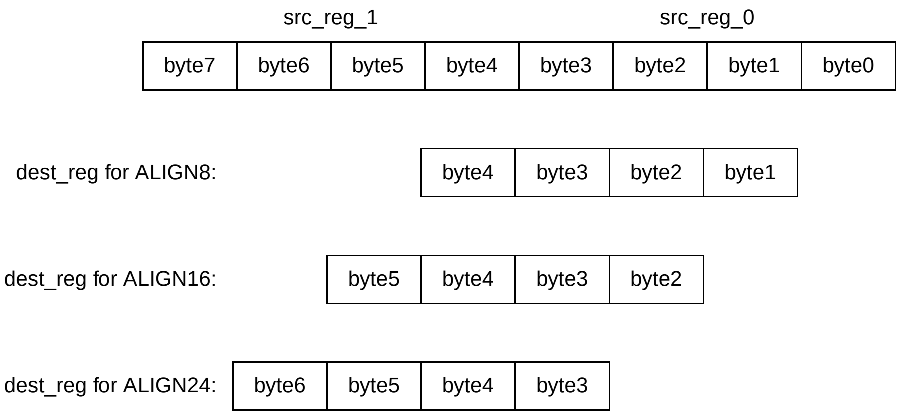
13.1.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

13.1.5 Functional 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.



The “ALIGN16” version performs the same operation as the Vector Pack instruction using the dest_reg = PACK (Dreg_lo, Dreg_hi) syntax.

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 so explicitly modifies that source register.

13.1.6 Flags Affected

None

13.1.7 Required Mode

User & Supervisor

13.1.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

13.1.9 Example

```
                            // If r3 = 0xABCD 1234 and r4 = 0xBEEF DEAD, then . . .
r0 = align8 (r3, r4) ;      // . . . produces r0 = 0x34BE EFDE,
r0 = align16 (r3, r4) ;     // . . . produces r0 = 0x1234 BEEF, and
r0 = align24 (r3, r4) ;     // . . . produces r0 = 0xCD12 34BE,
```

13.1.10 Also See

Vector Pack

13.1.11 Special Applications

13.2 Disable Alignment Exception for Load

13.2.1 General Form

DISALGNEXCPT

13.2.2 Syntax

```
DISALGNEXCPT ;                    // (b)
```

13.2.3 Syntax Terminology

13.2.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

13.2.5 Functional 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 2 LSB's 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.

13.2.6 Flags Affected

None

13.2.7 Required Mode

User & Supervisor

13.2.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, "Issuing Parallel Instructions."

13.2.9 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 */
```

13.2.10 Also See

Any Quad 8-Bit instructions, Section 13.1, “Byte Align,” on page 13-2

13.2.11 Special Applications

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 16 8-bit operands, four D-registers worth, to be preloaded with operand data. The operand data is 8-bit 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.

13.3 Dual 16-Bit Add / Clip

13.3.1 General Form

dest_reg = BYTEOP3P (src_reg_0, src_reg_1) (LO)

dest_reg = BYTEOP3P (src_reg_0, src_reg_1) (HI)

dest_reg = BYTEOP3P (src_reg_0, src_reg_1) (LO, R)

dest_reg = BYTEOP3P (src_reg_0, src_reg_1) (HI, R)

13.3.2 Syntax

```
// forward byte order operands
Dreg = BYTEOP3P (Dreg_pair,Dreg_pair) (LO)  ; // sum into low bytes (b)
Dreg = BYTEOP3P (Dreg_pair,Dreg_pair) (HI) ; // sum into high bytes (b)
```

```
// reverse byte order operands
Dreg = BYTEOP3P (Dreg_pair,Dreg_pair) (LO, R) ; // sum into low bytes (b)
Dreg = BYTEOP3P (Dreg_pair,Dreg_pair) (HI, R) ; // sum into high bytes (b)
```

13.3.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_pair: R1:0, R3:2, only

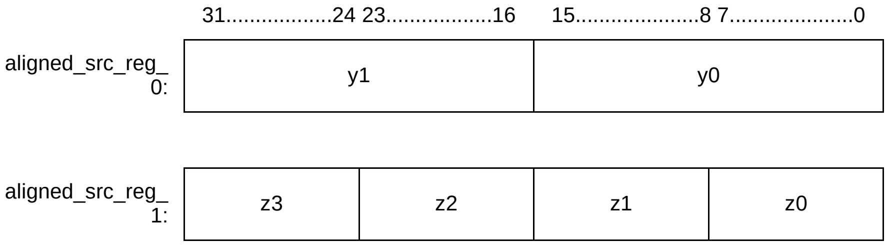
13.3.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

13.3.5 Functional 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 – 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 below.

Assuming the source registers contain:



…the versions that load the result into the lower byte – “(LO)” – produce:

	31................24	23................16	15..................8	7....................0
dest_reg:	0 0	y1 + z3 clipped to 8 bits	0 0	y0 + z1 clipped to 8 bits

…and the versions that load the result into the higher byte – “(HI)” – produce:

	31................24	23................16	15..................8	7....................0
dest_reg:	y1 + z2 clipped to 8 bits	0 0	y0 + z0 clipped to 8 bits	0 0

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.

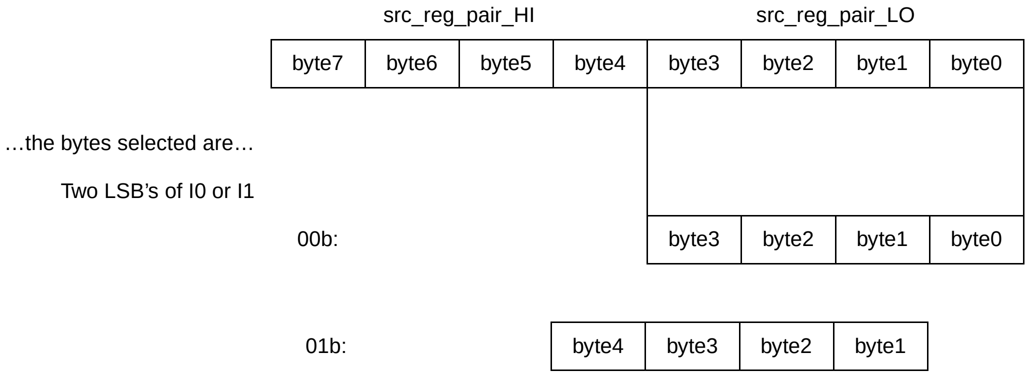
The only valid input source register pairs are R1:0 and R3:2.

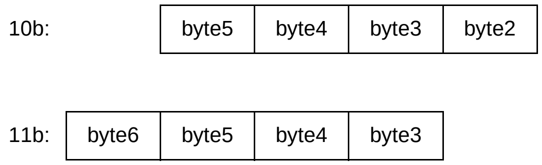
This instruction provides byte-alignment directly in the source register pairs R1:0 and R3:2 based on the I0 and I1 registers.

- The two LSB’s of the I0 register determine the byte-alignment for source register pair R1:0.
- The two LSB’s of the I1 register determine the byte-alignment for source register pair R3:2.

The relationship between the I-register bits and the byte-alignment is illustrated below.

In the default source order case (i.e., not the (– , R) syntax), assuming a source register pair contains the following:



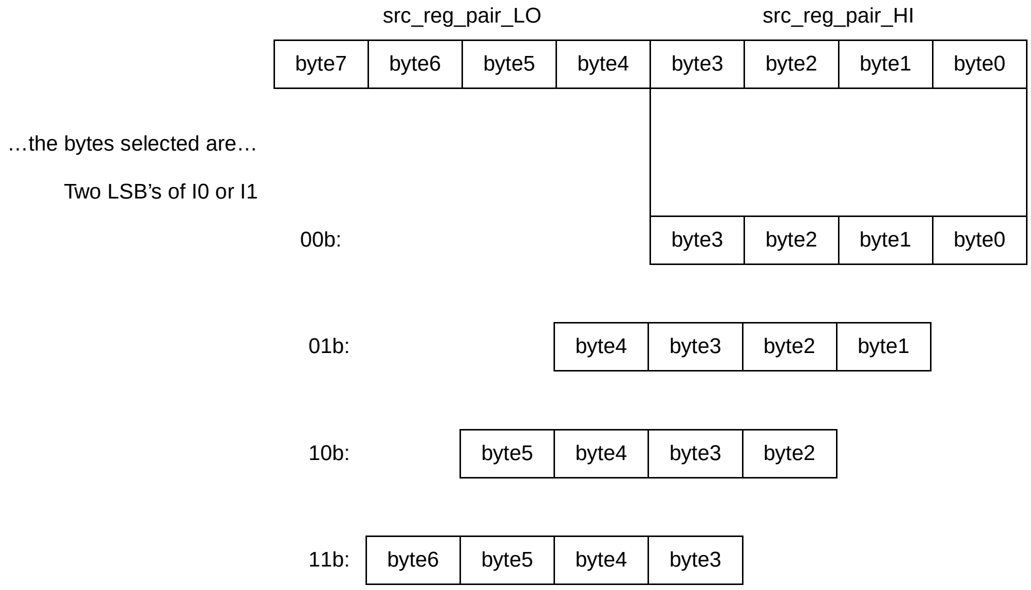


This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

13.3.6 Options

The (– , R) syntax reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of re-loading 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 (i.e., using the (– , R) syntax), the only difference is that *the source registers swap places* within the register pair in their byte-ordering. Assuming a source register pair contains the following:



13.3.7 Flags Affected

None

13.3.8 Required Mode

User & Supervisor

13.3.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

13.3.10 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) ;
```

13.3.11 Also See

Quad 8-Bit Add

13.3.12 Special Applications

This instruction is primarily intended for video motion compensation algorithms. It supports the addition of the residual to a video pixel value, followed by unsigned byte saturation.

13.4 Dual 16-Bit Accumulator Extraction with Addition

13.4.1 General Form

dest_reg_1 = A1.L + A1.H, dest_reg_0 = A0.L + A0.H

13.4.2 Syntax

Dreg = A1.L + A1.H, Dreg = A0.L + A0.H ; // (b)

13.4.3 Syntax Terminology

Dreg: R0, ..., R7

13.4.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

13.4.5 Functional Description

The Dual 16-Bit Accumulator Extraction with Addition instruction adds together the upper half-words (bits 31 to 16) and lower half-words (bits 15 to 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.

13.4.6 Flags Affected

None

13.4.7 Required Mode

User & Supervisor

13.4.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

13.4.9 Example

```
r4=a1.l+a1.h, r7=a0.l+a0.h ;
```

13.4.10 Also See

Quad 8-Bit Subtract-Absolute-Accumulate

13.4.11 Special Applications

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.

13.5 Quad 8-Bit Add

13.5.1 General Form

(dest_reg_1, dest_reg_0) = BYTEOP16P (src_reg_0, src_reg_1)

(dest_reg_1, dest_reg_0) = BYTEOP16P (src_reg_0, src_reg_1) (R)

13.5.2 Syntax

```
// forward byte order operands
( Dreg , Dreg ) = BYTEOP16P ( Dreg_pair , Dreg_pair ) ; // (b)
// reverse byte order operands
( Dreg , Dreg ) = BYTEOP16P ( Dreg_pair , Dreg_pair ) (R) ; // (b)
```

13.5.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_pair: R1:0, R3:2, only

13.5.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

13.5.5 Functional Description

The Quad 8-Bit Add instruction adds two unsigned quad byte number sets byte-wise, adjusting for byte-alignment. Load the byte-wise results as 16-bit, zero-extended, half-words in two destination registers, as shown below.

	31................24	23................16	15..................8	7....................0
aligned_src_reg_0:	y3	y2	y1	y0
aligned_src_reg_1:	z3	z2	z1	z0

	31..16	15..0
dest_reg_0:	y1 + z1	y0 + z0
dest_reg_1:	y3 + z3	y2 + z2

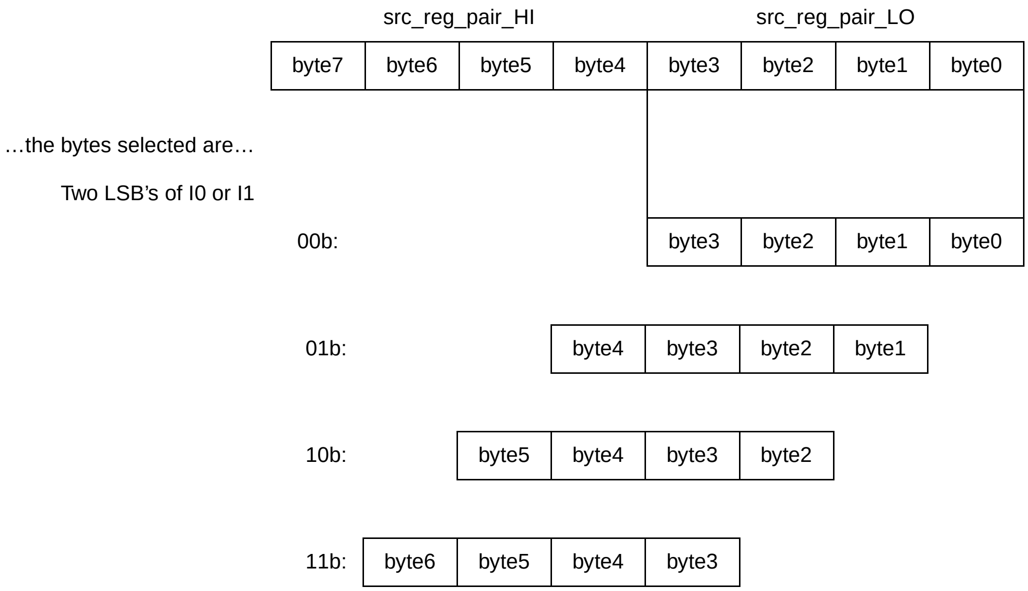
The only valid input source register pairs are R1:0 and R3:2.

The Quad 8-Bit Add instruction provides byte-alignment directly in the source register pairs R1:0 and R3:2 based on index registers I0 and I1.

- The two LSB's of the I0 register determine the byte-alignment for source register pair R1:0.
- The two LSB's of the I1 register determine the byte-alignment for source register pair R3:2.

The relationship between the I-register bits and the byte-alignment is illustrated below.

In the default source order case (i.e., not the (R) syntax), assuming that a source register pair contains the following:

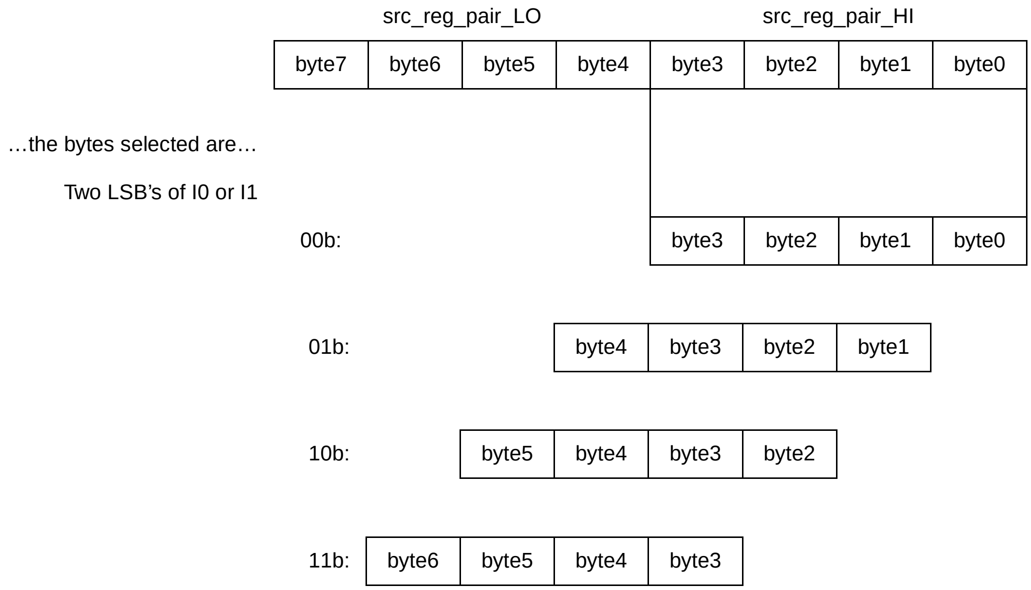


This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

13.5.6 Options

The (R) syntax reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of re-loading 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 (i.e., using the (R) syntax), the only difference is that the source registers swap places within the register pair in their byte-ordering. Assuming that a source register pair contains the following:



The mnemonic derives its name from the fact that the operands are bytes, the result is 16-bit, and the arithmetic operation is "plus" for addition.

13.5.7 Flags Affected

None

13.5.8 Required Mode

User & Supervisor

13.5.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, "Issuing Parallel Instructions."

13.5.10 Example

```
(r1,r2)= byteop16p (r3:2,r1:0);
(r1,r2)= byteop16p (r3:2,r1:0) (r);
```

13.5.11 Also See

Quad 8-Bit Subtract

13.5.12 Special Applications

This instruction provides packed data arithmetic typical of video and image processing applications.

13.6 Quad 8-Bit Average – Byte

13.6.1 General Form

dest_reg = BYTEOP1P (src_reg_0, src_reg_1)

dest_reg = BYTEOP1P (src_reg_0, src_reg_1) (T)

dest_reg = BYTEOP1P (src_reg_0, src_reg_1) (R)

dest_reg = BYTEOP1P (src_reg_0, src_reg_1) (T, R)

13.6.2 Syntax

```
                                                 // forward byte order operands
Dreg = BYTEOP1P (Dreg_pair,Dreg_pair) ;          // (b)
Dreg = BYTEOP1P (Dreg_pair,Dreg_pair) (T) ;      // truncated (b)
                                                 // reverse byte order operands
Dreg = BYTEOP1P (Dreg_pair,Dreg_pair) (R) ;      // (b)
Dreg = BYTEOP1P (Dreg_pair,Dreg_pair) (T, R) ;   // truncated (b)
```

13.6.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_pair: R1:0, R3:2, only

13.6.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

13.6.5 Functional 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 below.

	31................24	23................16	15..................8	7....................0
aligned_src_reg_0:	y3	y2	y1	y0
aligned_src_reg_1:	z3	z2	z1	z0

	31................24	23................16	15.................8	7...................0
dest_reg:	avg(y3, z3)	avg(y2, z2)	avg(y1, z1)	avg(y0, z0)

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 Section 1.5.6, “Rounding and Truncating,” on page 1-7 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.

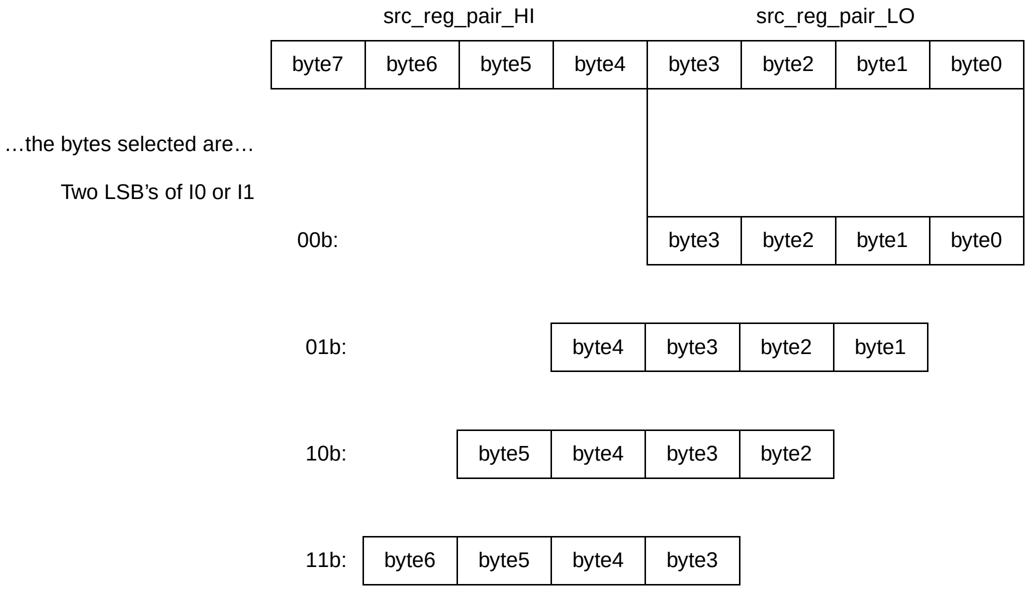
The only valid input source register pairs are R1:0 and R3:2.

This instruction provides byte-alignment directly in the source register pairs R1:0 and R3:2 based on the I0 and I1 registers:

- The two LSB’s of the I0 register determine the byte-alignment for source register pair R1:0.
- The two LSB’s of the I1 register determine the byte-alignment for source register pair R3:2.

The relationship between the I-register bits and the byte-alignment is illustrated below.

In the default source order case (i.e., 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.

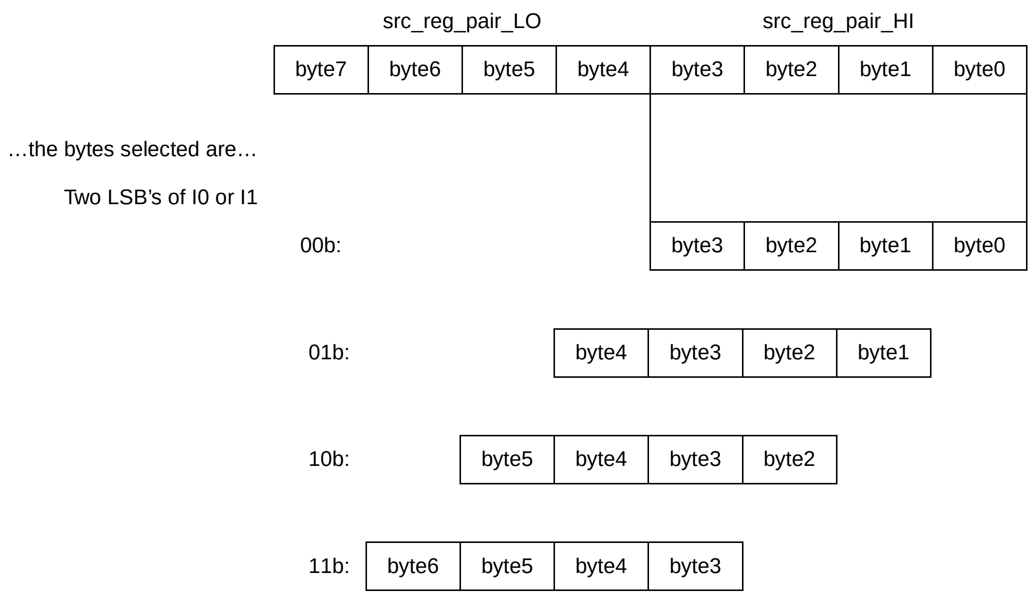
13.6.6 Options

The Quad 8-Bit Average – Byte instruction supports the following options:

Table 13-1. Options for Quad 8-Bit Average – Byte

Option	Description
Default	Round up the arithmetic mean.
(T)	Truncates the arithmetic mean.
(R)	Reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of re-loading 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 (i.e., using the (R) syntax), the only difference is that *the source registers swap places* within the register pair in their byte-ordering. Assuming that a source register pair contains the following:



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.

13.6.7 Flags Affected

None

13.6.8 Required Mode

User & Supervisor

13.6.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

13.6.10 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) ;
```

13.6.11 Also See

Quad 8-Bit Add

13.6.12 Special Applications

This instruction supports binary interpolation used in fractional motion search and motion compenstion algorithms.

13.7 Quad 8-Bit Average – Half-Word

13.7.1 General Form

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (RNDL)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (RNDH)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (TL)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (TH)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (RNDL, R)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (RNDH, R)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (TL, R)

dest_reg = BYTEOP2P (src_reg_0, src_reg_1) (TH, R)

13.7.2 Syntax

```
                                                 // forward byte order operands
Dreg = BYTEOP2P (Dreg_pair,Dreg_pair) (RNDL)  ;   // round into low bytes (b)
Dreg = BYTEOP2P (Dreg_pair,Dreg_pair) (RNDH)  ;   // round into high bytes (b)
Dreg = BYTEOP2P (Dreg_pair,Dreg_pair) (TL) ;      // truncate into low bytes (b)
Dreg = BYTEOP2P (Dreg_pair,Dreg_pair) (TH) ;      // truncate into high bytes (b)
                                                 // reverse byte order operands
Dreg = BYTEOP2P (Dreg_pair,Dreg_pair) (RNDL, R) ; // round into low bytes (b)
Dreg = BYTEOP2P (Dreg_pair,Dreg_pair) (RNDH, R) ; // round into high bytes (b)
Dreg = BYTEOP2P (Dreg_pair,Dreg_pair) (TL, R) ;   // truncate into low bytes (b)
Dreg = BYTEOP2P (Dreg_pair,Dreg_pair) (TH, R) ;   // truncate into high bytes (b)
```

13.7.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_pair: R1:0, R3:2, only

13.7.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

13.7.5 Functional 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 below.

Assuming the source registers contain…

	31…………….24	23…………….16	15……………..8	7………………..0
aligned_src_reg_0:	y3	y2	y1	y0
aligned_src_reg_1:	z3	z2	z1	z0

…the versions that load the result into the lower byte – RND**L** and T**L** – produce…

	31…………….24	23…………….16	15……………..8	7………………..0
dest_reg:	0 0	avg(y3, y2, z3, z2)	0 0	avg(y1, y0, z1, z0)

…and the versions that load the result into the higher byte – RND**H** and T**H** – produce…

	31…………….24	23…………….16	15……………..8	7………………..0
dest_reg:	avg(y3, y2, z3, z2)	0 0	avg(y1, y0, z1, z0)	0 0

In either case, the unused bytes in the destination register 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.

See Section 1.5.6, “Rounding and Truncating,” on page 1-7 for a description of unbiased rounding and truncating behavior.

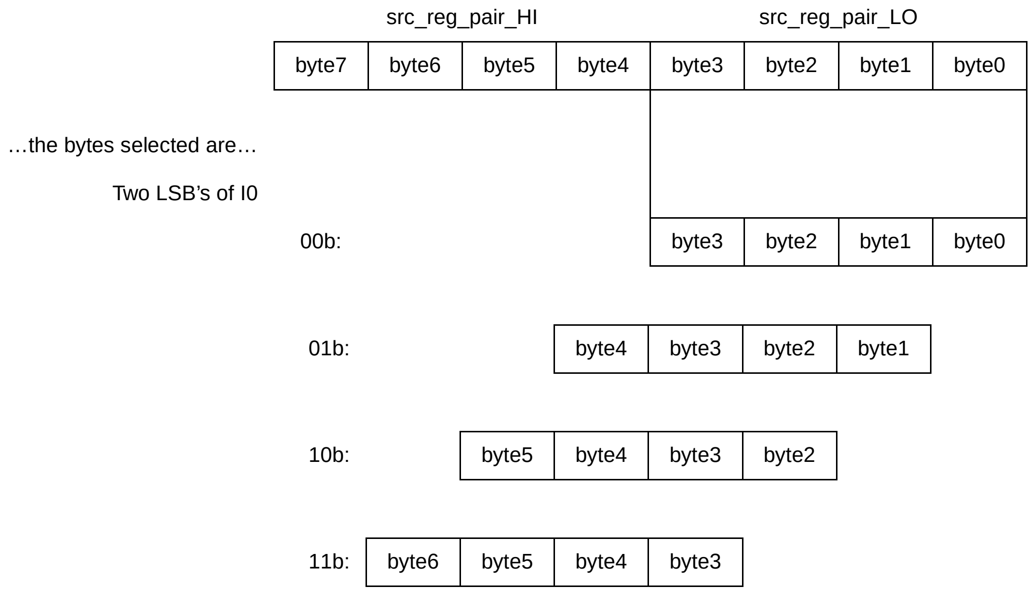
The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

The only valid input source register pairs are R1:0 and R3:2.

The Quad 8-Bit Average – Half-Word instruction provides byte-alignment directly in the source register pairs R1:0 and 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 illustrated next.

In the default source order case (i.e., 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.

13.7.6 Options

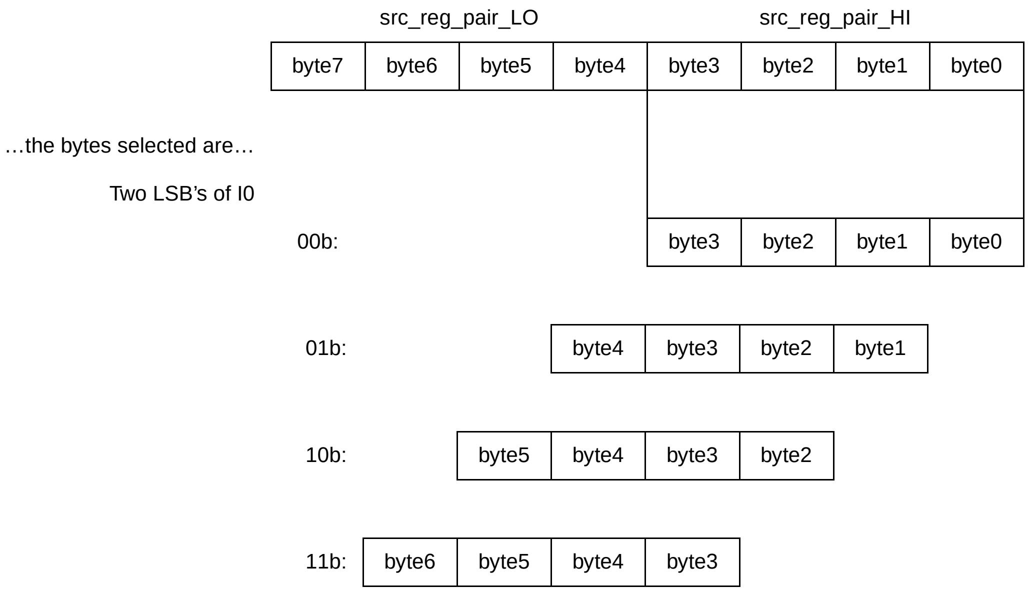
The Quad 8-Bit Average – Half-Word instruction supports the following options:

Table 13-2. 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 re-loading 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 (i.e., using the (R) syntax), the only difference is that *the source registers swap places* within the register pair in their byte-ordering. Assuming a source register pair contains the following:



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.

13.7.7 Flags Affected

None

13.7.8 Required Mode

User & Supervisor

13.7.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

13.7.10 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) ;
```

13.7.11 Also See

Quad 8-Bit Average – Byte

13.7.12 Special Applications

This instruction supports binary interpolation used in fractional motion search and motion compenstion algorithms.

13.8 Quad 8-Bit Pack

13.8.1 General Form

dest_reg = BYTEPACK (src_reg_0, src_reg_1)

13.8.2 Syntax

Dreg = BYTEPACK (Dreg , Dreg) ;   // (b)

13.8.3 Syntax Terminology

Dreg: R0, ..., R7

13.8.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

13.8.5 Functional 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.

	31................24	23................16	15..................8	7...................0
src_reg_0:		byte1		byte0
src_reg_1:		byte3		byte2
dest_reg:	byte3	byte2	byte1	byte0

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

13.8.6 Flags Affected

None

13.8.7 Required Mode

User & Supervisor

13.8.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

13.8.9 Example

r2 = bytepack (r4,r5);

Assuming…
R4 = 0xFEED FACE
R5 = 0xBEEF BADD

then this instruction returns…
R2 = 0xEFDD EDCE

13.8.10 Also See

Quad 8-Bit Unpack

13.8.11 Special Applications

13.9 Quad 8-Bit Subtract

13.9.1 General Form

(dest_reg_1, dest_reg_0) = BYTEOP16M (src_reg_0, src_reg_1)

(dest_reg_1, dest_reg_0) = BYTEOP16M (src_reg_0, src_reg_1) (R)

13.9.2 Syntax

// forward byte order operands
(Dreg , Dreg) = BYTEOP16M (Dreg_pair , Dreg_pair) ;   // (b)

// reverse byte order operands
(Dreg, Dreg) = BYTEOP16M (Dreg-pair, Dreg-pair) (R) ;   // (b)

13.9.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_pair: R1:0, R3:2, only

13.9.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

13.9.5 Functional Description

The Quad 8-Bit Subtract instruction subtracts two unsigned quad byte number sets byte-wise, adjusting for byte-alignment. The instruction loads the byte-wise results as sign-extended half-words in two destination registers, as shown below.

	31................24	23................16	15..................8	7....................0
aligned_src_reg_0:	y3	y2	y1	y0
aligned_src_reg_1:	z3	z2	z1	z0

	31..16	15..0
dest_reg_0:	y1 - z1	y0 - z0
dest_reg_1:	y3 - z3	y2 - z2

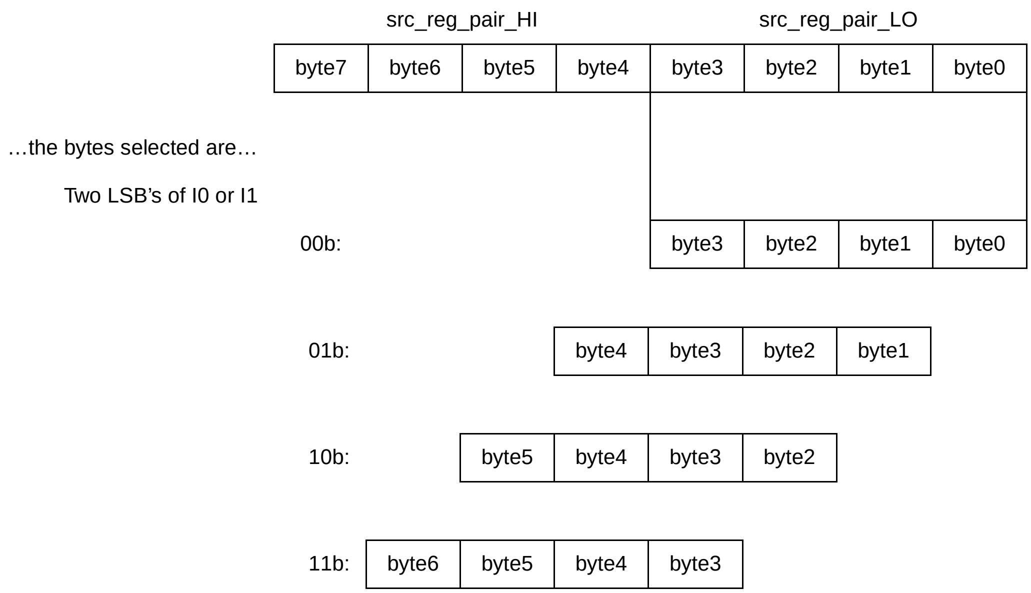
The only valid input source register pairs are R1:0 and R3:2.

The Quad 8-Bit Subtract instruction provides byte-alignment directly in the source register pairs R1:0 and R3:2 based on index registers I0 and I1.

- The two LSB’s of the I0 register determine the byte-alignment for source register pair R1:0.
- The two LSB’s of the I1 register determine the byte-alignment for source register pair R3:2.

The relationship between the I-register bits and the byte-alignment is illustrated below.

In the default source order case (i.e., not the (R) syntax), assuming that a source register pair contains the following:

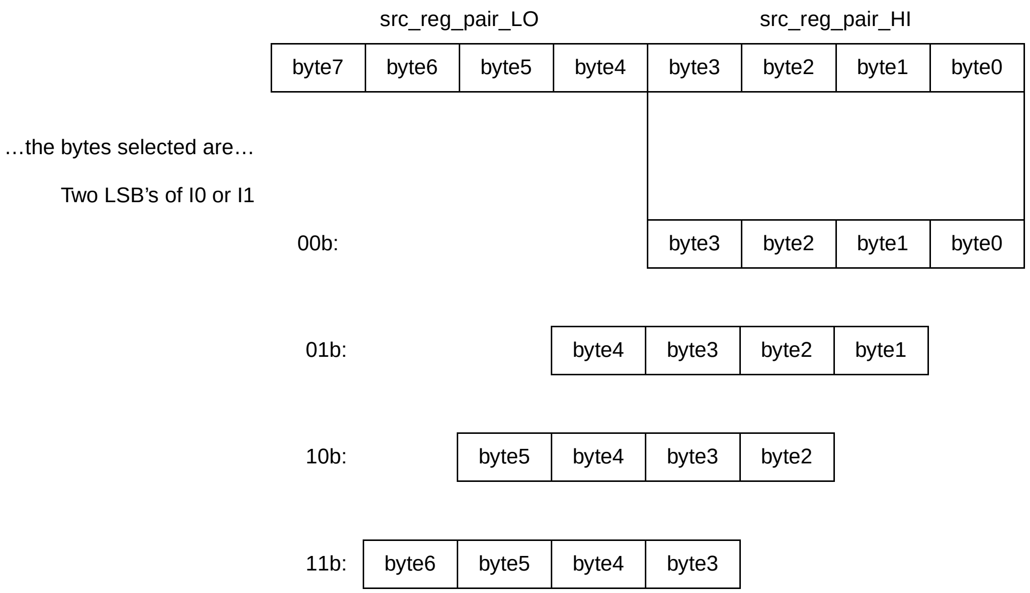


This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

13.9.6 Options

The (R) syntax reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of re-loading 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 (i.e., using the (R) syntax), the only difference is that the source registers swap places within the register pair in their byte-ordering. Assuming that a source register pair contains the following:



The mnemonic derives its name from the fact that the operands are bytes, the result is 16-bit, and the arithmetic operation is "minus" for subtraction.

13.9.7 Flags Affected

None

13.9.8 Required Mode

User & Supervisor

13.9.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, "Issuing Parallel Instructions."

13.9.10 Example

```
(r1,r2)= byteop16m (r3:2,r1:0);
(r1,r2)= byteop16m (r3:2,r1:0) (r);
```

13.9.11 Also See

Quad 8-Bit Add

13.9.12 Special Applications

This instruction provides packed data arithmetic typical of video and image processing applications.

13.10 Quad 8-Bit Subtract-Absolute-Accumulate

13.10.1 General Form

SAA (src_reg_0, src_reg_1)

SAA (src_reg_0, src_reg_1) (R)

13.10.2 Syntax

```
SAA ( Dreg_pair , Dreg_pair ) ;          // forward byte order operands (b)
SAA ( Dreg_pair , Dreg_pair ) (R) ;      // reverse byte order operands (b)
```

13.10.3 Syntax Terminology

Dreg_pair: R1:0, R3:2 (This instruction only supports register pairs R1:0 and R3:2.)

13.10.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

13.10.5 Functional 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.

No saturation is performed by this operation.

Only register pairs R1:0 and R3:2 are valid sources for this instruction.

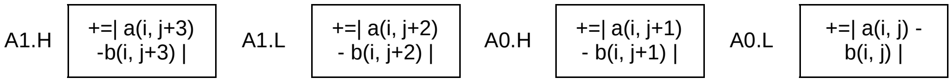
This instruction supports the following byte-wise sum of absolute difference (SAD) calculations:

$$SAD = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left|a(i, j) - b(i, j)\right|$$

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 below.

src_reg_0	a(i, j+3)	a(i, j+2)	a(i, j+1)	a(i, j)
src_reg_1	b(i, j+3)	b(i, j+2)	b(i, j+1)	b(i, j)

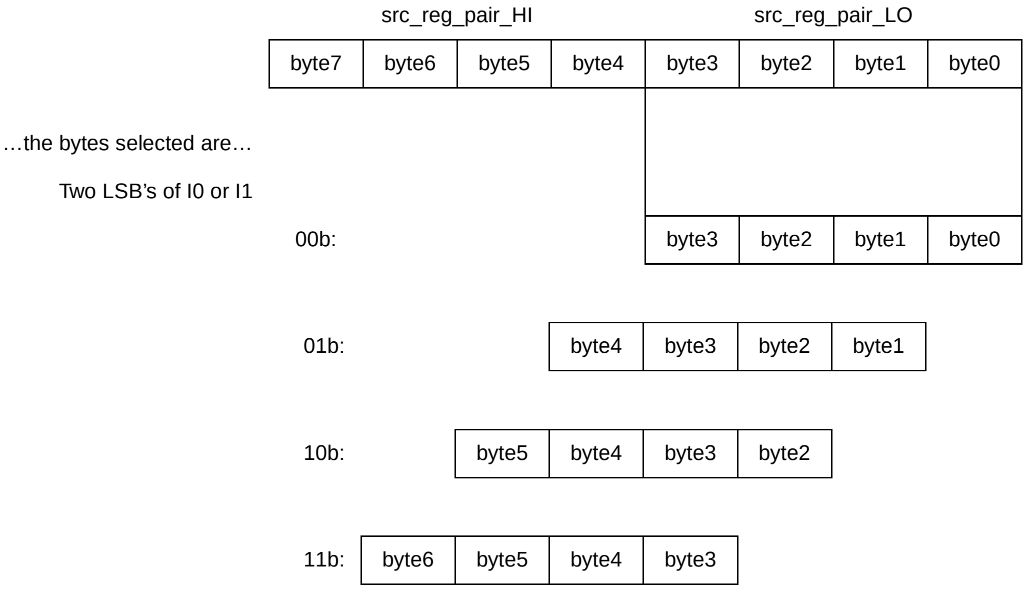

A1.H
+=| a(i, j+3) -b(i, j+3) |
A1.L
+=| a(i, j+2) - b(i, j+2) |
A0.H
+=| a(i, j+1) - b(i, j+1) |
A0.L
+=| a(i, j) - b(i, j) |

The only valid input source register pairs are R1:0 and R3:2. This instruction provides byte-alignment directly in the source register pairs R1:0 and R3:2 based on the I0 and I1 registers:

- The two LSB's of the I0 register determine the byte-alignment for source register pair src_reg_0.
- The two LSB's of the I1 register determine the byte-alignment for source register pair src_reg_1.

The relationship between the I-register bits and the byte-alignment is illustrated below.

In the default source order case (i.e., not the (R) syntax), assuming a source register pair contain:

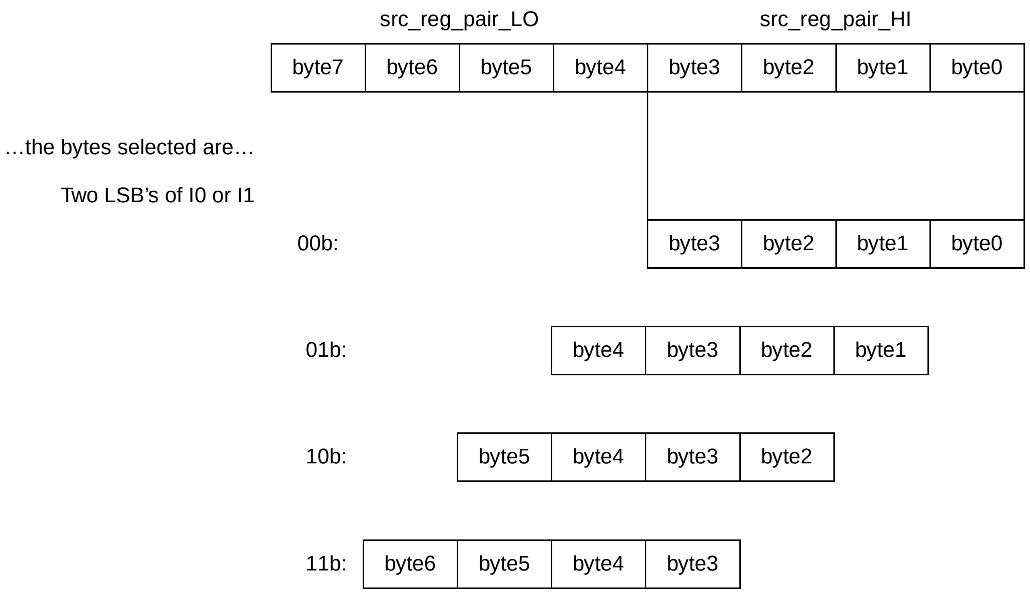


This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

13.10.6 Options

The (R) syntax reverses the order of the source registers within each pair. Typical high performance applications cannot afford the overhead of re-loading 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 following



The SAA instruction computes 12 pixel operations simultaneously – the 3-operation subtract-absolute-accumulate on 4 pairs of operand bytes in parallel.

13.10.7 Flags Affected

None

13.10.8 Required Mode

User & Supervisor

13.10.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

13.10.10 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
```

13.10.11 Also See

Disable Alignment Exception for Load, Load Data Register

13.10.12 Special Applications

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.

13.11 Quad 8-Bit Unpack

13.11.1 General Form

(dest_reg_1, dest_reg_0) = BYTEUNPACK src_reg_pair

(dest_reg_1, dest_reg_0) = BYTEUNPACK src_reg_pair (R)

13.11.2 Syntax

(Dreg , Dreg) = BYTEUNPACK Dreg_pair ; // (b)

(Dreg , Dreg) = BYTEUNPACK Dreg_pair (R) ; // reverse source order (b)

13.11.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_pair: R1:0, R3:2, only

13.11.4 Instruction Length

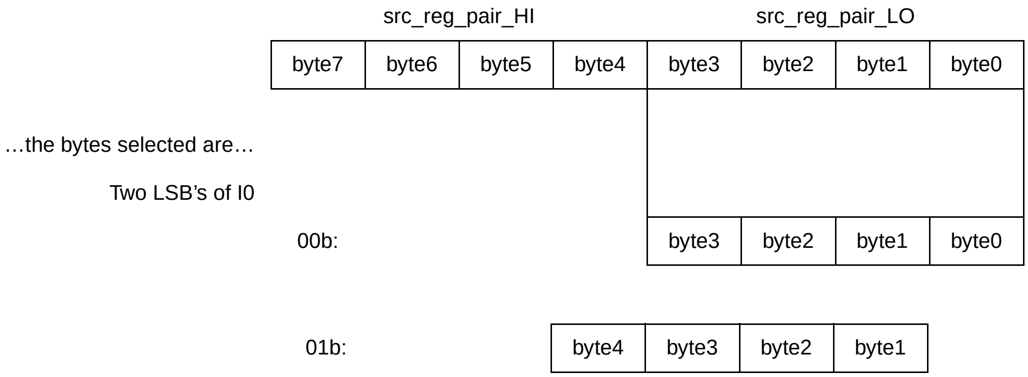
In the syntax, comment (b) identifies 32-bit instruction length.

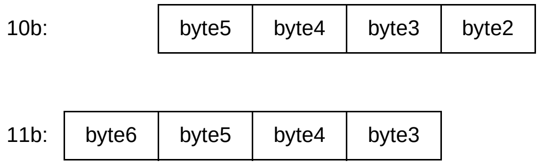
13.11.5 Functional Description

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 LSB’s of the I0 register determine the source byte-alignment, as illustrated below.

In the default source order case (i.e., not the (R) syntax), assuming that the source register pair contains the following:



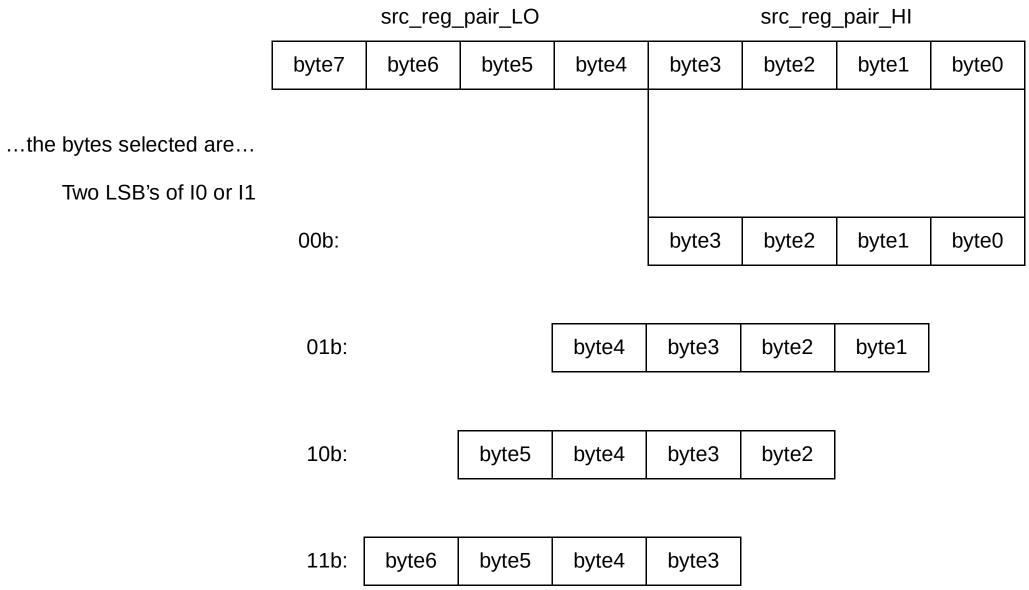


This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

13.11.6 Options

The (R) syntax reverses the order of the source registers within the pair. Typical high performance applications cannot afford the overhead of re-loading 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 (i.e., using the (R) syntax), the only difference is that *the source registers swap places* in their byte-ordering. Assuming the source register pair contains the following:



The four bytes, now byte aligned, are copied into the destination registers on half-word alignment, as shown below.

	31................24	23................16	15..................8	7....................0
Aligned bytes:	byte_D	byte_C	byte_B	byte_A

	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.

Misaligned access exceptions are disabled during this instruction.

13.11.7 Flags Affected

None

13.11.8 Required Mode

User & Supervisor

13.11.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, "Issuing Parallel Instructions."

13.11.10 Example

```
(r6,r5) = byteunpack r1:0  ;          // non-reversing sources
```

Assuming…

register I0's two LSB's = **00**b,
R1 = 0xFEED FACE
R0 = 0xBEEF BADD

then this instruction returns…

R6 = 0x00BE 00EF
R5 = 0x00BA 00DD

Assuming…

register I0's two LSB's = **01**b,
R1 = 0xFEED FACE
R0 = 0xBEEF BADD

then this instruction returns…

R6 = 0x00CE 00BE
R5 = 0x00EF 00BA

Assuming…
register I0’s two LSB’s = **10**b,
R1 = 0xFEED FACE
R0 = 0xBEEF BADD

then this instruction returns…
R6 = 0x00FA 00CE
R5 = 0x00BE 00EF

Assuming…
register I0’s two LSB’s = **11**b,
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
```

Assuming…
register I0’s two LSB’s = **00**b,
R1 = 0xFEED FACE
R0 = 0xBEEF BADD

then this instruction returns…
R6 = 0x00FE 00ED
R5 = 0x00FA 00CE

Assuming…
register I0’s two LSB’s = **01**b,
R1 = 0xFEED FACE
R0 = 0xBEEF BADD

then this instruction returns…
R6 = 0x00DD 00FE
R5 = 0x00ED 00FA

Assuming…
register I0’s two LSB’s = **10**b,
R1 = 0xFEED FACE
R0 = 0xBEEF BADD

then this instruction returns…
R6 = 0x00BA 00DD
R5 = 0x00FE 00ED

Assuming…
register I0’s two LSB’s = **11**b,

R1 = 0xFEED FACE
R0 = 0xBEEF BADD

then this instruction returns…
R6 = 0x00EF 00BA
R5 = 0x00DD 00FE

13.11.11 Also See

Quad 8-bit Pack

13.11.12 Special Applications

VECTOR OPERATIONS 14

Instruction Summary

This chapter discusses the instructions that control vector operations. Users can take advantage of these instructions to perform simultaneous operations on multiple 16-bit values, including add, subtract, multiply, shift, negate, pack and search. Compare-Select and Add-On-Sign are also included in this section.

14.1 Add on Sign

14.1.1 General Form

dest_hi = dest_lo = SIGN (src0_hi) * src1_hi + SIGN (src0_lo) * src1_lo

14.1.2 Syntax

Dreg_hi = Dreg_lo = SIGN (Dreg_hi) * Dreg_hi + SIGN (Dreg_lo) * Dreg_lo ; // (b)

REGISTER CONSISTANCY

The destination registers dest_hi and dest_lo must be halves of the same data register. Similarly, src0_hi and src0_lo must be halves of the same register and src1_hi and src1_lo must be halves of the same register.

14.1.3 Syntax Terminology

Dreg_hi: R0.H, ..., R7.H

Dreg_lo: R0.L, ..., R7.L

14.1.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

14.1.5 Functional Description

The Add on Sign instruction performs a two-step function, as follows:

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 below. *Y* is the number in src0, and *Z* is the number in src1. The numbers in src0 and src1 may be positive or negative.

SRC0	SRC1	Sign-Adjusted SRC1
+Y	+Z	+Z
+Y	–Z	–Z
–Y	+Z	–Z
–Y	–Z	+Z

Note that the result always bears the magnitude of Z with only the sign affected.

2. Then, 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 illustration, below.

   Assuming the source registers contain…

	31................24 23................16	15..................8 7....................0
src0:	a1	a0
src1:	b1	b0

…the destination register contains…

dest:	(sign_adjusted_b1) + (sign_adjusted_b0)	(sign_adjusted_b1) + (sign_adjusted_b0)

The sum is not saturated if the addition exceeds 16 bits.

14.1.6 Flags Affected

None.

14.1.7 Required Mode

User & Supervisor

14.1.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

14.1.9 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

14.1.10 Also See

14.1.11 Special Applications

Use the Sum on Sign instruction to compute the branch metric used by each Viterbi Butterfly.

14.2 Compare-Select (VIT_MAX)

14.2.1 General Form

```
dest_reg = VIT_MAX ( src_reg_0, src_reg_1 ) (ASL)
dest_reg = VIT_MAX ( src_reg_0, src_reg_1 ) (ASR)
dest_reg_lo = VIT_MAX ( src_reg ) (ASL)
dest_reg_lo = VIT_MAX ( src_reg ) (ASR)
```

14.2.2 Syntax

DUAL 16-BIT OPERATION

```
Dreg = VIT_MAX ( Dreg , Dreg ) (ASL) ;     // shift history bits left (b)
Dreg = VIT_MAX ( Dreg , Dreg ) (ASR) ;     // shift history bits right (b)
```

SINGLE 16-BIT OPERATION

```
Dreg_lo = VIT_MAX ( Dreg ) (ASL) ;         // shift history bits left (b)
Dreg_lo = VIT_MAX ( Dreg ) (ASR) ;         // shift history bits right (b)
```

14.2.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo: R0.L, ..., R7.L

14.2.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

14.2.5 Functional 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.

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.

The Accumulator extension bits (bits 39:32) must be cleared before executing this instruction.

This operation is illustrated, below.

DUAL 16-BIT OPERAND BEHAVIOR

If the source registers contain the following:

	31.................24 23.................16	15.....................8 7.....................0
src_reg_0	y1	y0
src_reg_1	z1	z0

…the destination register will contain…

dest_reg	Maximum, y1 or y0	Maximum, z1 or z0

The ASL version shifts A0 left two bit positions and appends two LSB's to indicate the source of each maximum.

	A0.X	A0.W
A0	00000000	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX**BB**

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 MSB's to indicate the source of each maximum.

	A0.X	A0.W
A0	00000000	**BB**XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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.

DUAL 16-BIT OPERAND BEHAVIOR

If the dual source register contains the following:

	31...................24 23.................16	15....................8 7.....................0
src_reg	y1	y0

…the dual destination register will contain…

dest_reg_lo		Maximum, y1 or y0

The ASL version shifts A0 left one bit position and appends an LSB to indicate the source of the maximum.

	A0.X	A0.W
A0	00000000	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX**B**

Conversely, the ASR version shifts A0 right one bit position and appends an MSB to indicate the source of the maximum.

	A0.X	A0.W
A0	00000000	**B**XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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 2'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.

14.2.6 Flags Affected

None.

14.2.7 Required Mode

User & Supervisor

14.2.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

14.2.9 Example

```
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
```

```
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

```
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

```
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 7FFF FFFF

14.2.10 Also See

Maximum (under “Arithmetic Operations”)

14.2.11 Special Applications

The Compare-Select (VIT_MAX) instruction is 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.

14.3 Vector Absolute Value

14.3.1 General Form

dest_reg = ABS source_reg (V)

14.3.2 Syntax

Dreg = ABS Dreg (V) ;   // (b)

14.3.3 Syntax Terminology

Dreg: R0, ..., R7

14.3.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

14.3.5 Functional 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.

For example, if the source register contains this:

	31................24 23...............16	15..................8 7..................0
src_reg:	x.h	x.l

…the destination register contains this:

	31................24 23...............16	15..................8 7..................0
dest_reg:	| x.h|	| x.l |

This instruction saturates the result.

14.3.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if either or both result is zero; cleared if both are non-zero.
- AN is cleared.
- V is set if either or both result saturates; cleared if both are no saturation.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

14.3.7 Required Mode

User & Supervisor

14.3.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

14.3.9 Example

```
            // If r1 = 0xFFFF 7FFF, then . . .
r3 = abs r1 (v);
            // . . . produces 0x0001 7FFF
```

14.3.10 Also See

Absolute Value (in “Arithmetic Operations” chapter)

14.3.11 Special Applications

14.4 Vector Add / Subtract

14.4.1 General Form

dest = src_reg_0+ |+ src_reg_1

dest = src_reg_0 –|+ src_reg_1

dest = src_reg_0 +|– src_reg_1

dest = src_reg_0 –|– src_reg_1

dest_0 = src_reg_0 +|+ src_reg_1, dest_1 = src_reg_0 –|– src_reg_1

dest_0 = src_reg_0 +|– src_reg_1, dest_1 = src_reg_0 –|+ src_reg_1

dest_0 = src_reg_0 + src_reg_1, dest_1 = src_reg_0 – src_reg_1

dest_0 = A1 + A0, dest_1 = A1 – A0

dest_0 = A0 + A1, dest_1 = A0 – A1

14.4.2 Syntax

DUAL 16-BIT OPERATIONS

```
Dreg = Dreg +|+ Dreg   (opt_mode0) ;                 // add | add (b)
Dreg = Dreg –|+ Dreg   (opt_mode0) ;                 // subtract | add (b)
Dreg = Dreg +|– Dreg   (opt_mode0) ;                 // add | subtract (b)
Dreg = Dreg –|– Dreg   (opt_mode0) ;                 // subtract | subtract (b)
```

QUAD 16-BIT OPERATIONS

```
Dreg = Dreg +|+ Dreg,   Dreg = Dreg –|– Dreg   (opt_mode1, opt_mode2) ;
                  /* add | add, subtract | subtract; the set of source registers must be the
                  same for each operation (b) */

Dreg = Dreg +|– Dreg,   Dreg = Dreg –|+ Dreg   (opt_mode1, opt_mode2) ;
                  /* add | subtract, subtract | add; the set of source registers must be the
                  same for each operation (b) */
```

DUAL 32-BIT OPERATIONS

```
Dreg = Dreg + Dreg,   Dreg = Dreg – Dreg   (opt_mode1) ;
                  /* add, subtract; the set of source registers must be the
                  same for each operation (b) */
```

DUAL 40-BIT ACCUMULATOR OPERATIONS

```
Dreg = A1 + A0,   Dreg = A1 – A0   (opt_mode1) ;      /* add, subtract Accumulators;
                                                      subtract A0 from A1 (b) */
```

```
Dreg = A0 + A1,   Dreg = A0 – A1   (opt_mode1)        /* add, subtract Accumulators;
                                                        subtract A1 from A0 (b) */
```

14.4.3 Syntax Terminology

Dreg: R0, ..., R7

opt_mode0: optional (S), (CO), or (SCO)

opt_mode1: optional (S)

opt_mode2: optional (ASR), or (ASL)

14.4.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

14.4.5 Functional 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.

14.4.6 Options

The Vector Add / Subtract instruction provides three option modes:

- opt_mode0 supports the Dual 16-Bit Operations versions of this instruction.
- opt_mode1 supports the Quad 16-Bit Operations, 32-bit and 40-bit operations.
- opt_mode2 supports the Quad 16-Bit Operations versions of this instruction.

Table 14.6 describes the options that the three opt_modes support.

Table 14-1. Options for Opt_Mode 0

Mode	Option	Description
opt-mode0	S	saturate the results at 16 bits
	CO	cross option. Swap the order of the results in the destination register.
	SCO	saturate and cross option. Combination of (S) and (CO) options.
opt_mode1	S	saturate the results at 16 or 32 bits, depending on the operand size
opt_mode2	ASR	arithmetic shift right. Halve the result (divide by 2) before storing in the destination register. If specified with the S (saturation) flag in Quad 16-Bit Operand versions of this instruction, the scaling is performed before saturation.
	ASL	arithmetic shift left. Double the result (multiply by 2, truncated) before storing in the destination register. If specified with the S (saturation) flag in Quad 16-Bit Operand versions of this instruction, the scaling is performed before saturation.

The options shown for opt_mode2 are scaling options.

14.4.7 Flags Affected

This instruction affects the following flags:

- AZ is set if any results are zero; cleared if all are non-zero.
- AN is set if any results are negative; cleared if all non-negative.
- AC0 is set if the operation on the right-hand of the instruction pair generates a carry; cleared if no carry.
- AC1 is set if the operation on the left-hand of the instruction pair generates a carry; cleared if no carry.
- V is set if any results overflow; cleared if none overflows.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

14.4.8 Required Mode

User & Supervisor

14.4.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, "Issuing Parallel Instructions."

14.4.10 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 */

r2=r0+r1, r3=r0-r1 ;                 // 32-bit operations

r2=r0+r1, r3=r0-r1(s) ;              // dual 32-bit operations with saturation

r4=a1+a0, r6=a1-a0 ;                 /* dual 40-bit Accumulator operations, A0
                                     subtracted from A1 */

r4=a1+a0, r6=a1-a0(s ) ;             /* dual 40-bit Accumulator operations with
                                     saturation, A1 subtracted from A0 */
```

14.4.11 Also See

Add (in the “Arithmetic Operations” chapter), Subtract (in the “Arithmetic Operations” chapter)

14.4.12 Special Applications

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
```

14.5 Vector Arithmetic Shift

14.5.1 General Form

dest_reg = src_reg >>> shift_magnitude (V)

dest_reg = ASHIFT src_reg BY shift_magnitude (V)

14.5.2 Syntax

CONSTANT SHIFT MAGNITUDE

Dreg = Dreg >>> uimm4 (V) ; // arithmetic shift right, immediate (b)

Dreg = Dreg << uimm4 (V,S) ; // arithmetic shift left, immediate with saturation (b)

REGISTERED SHIFT MAGNITUDE

Dreg = ASHIFT Dreg BY Dreg_lo (V) ; // arithmetic shift (b)

Dreg = ASHIFT Dreg BY Dreg_lo (V, S) ; // arithmetic shift with saturation (b)

ARITHMETIC LEFT SHIFT IMMEDIATE

There is no syntax specific to a vector arithmetic left shift immediate instruction. Use the Vector Logical Shift syntax for vector left shifting, which accomplishes the same function for sign-extended numbers in number-normalizing routines. See '">>>" SYNTAX' notes for caveats.

14.5.3 Syntax Terminology

Dreg: R0, ..., R7

Dreg_lo: R0.L, ..., R7.L

uimm4: unsigned 4-bit field, with a range of 0 through 15

14.5.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

14.5.5 Functional Description

The Vector Arithmetic Shift instruction arithmetically shifts a pair of half-word registered numbers a specified distance and direction. Though the two half-word registers are shifted at the same time, the two numbers are kept separate.

Arithmetic right shifts preserve the sign of the pre-shifted value. The sign bit value back-fills the left-most bit position vacated by the arithmetic right shift. For positive numbers, this behavior is equivalent to the logical right shift for unsigned numbers.

Only arithmetic right shifts are supported. Left shifts are performed as logical left shifts that may not preserve the sign of the original number. In the default case – without the optional saturation option – numbers can be left shifted so far that all the sign bits overflow and are lost. However, when the saturation option is enabled, a left shift that would otherwise shift non-sign bits off the left side saturates to the maximum positive or negative value instead. So, with saturation enabled, the result always keeps the same sign as the original number.

See Section 1.5.5, "Saturation," on page 1-6 for a description of saturation behavior.

">>>" and "<<" SYNTAX

The two half-word registers in dest_reg are right shifted by the number of places specified by shift_magnitude and the result stored into dest_reg. The data is always a pair of 16-bit half-registers. Valid shift_magnitude values are 0 – 15.

"ASHIFT" SYNTAX

Both half-word registers in src_reg are shifted by the number of places prescribed in shift_magnitude and the result stored into dest_reg.

The the sign of the shift magnitude determines the direction of the shift for the ASHIFT versions, as follows:

- Positive shift magnitudes without the saturation (– , S) flag produce LOGICAL LEFT shifts.
- Positive shift magnitudes with the saturation (– , S) flag produce ARITHMETIC LEFT shifts
- Negative shift magnitudes produce ARITHMETIC RIGHT shifts.

In essence, the magnitude is the power of 2 multiplied by the src_reg number. Positive magnitudes cause multiplication ($N \times 2^n$) whereas negative magnitudes produce division ($N \times 2^{-n}$ or $N / 2^n$).

The dest_reg and src_reg are both pairs of 16-bit half-registers. Saturation of the result is optional.

Valid shift magnitudes for 16-bit src_reg are –16 through +15, zero included. If a number larger than these is supplied, the instruction masks and ignores the more significant bits.

This instruction does not implicitly modify the src_reg values. Optionally, dest_reg can be the same D-register as src_reg. Using the same D-register for the dest_reg and the src_reg explicitly modifies the source register at your discretion.

14.5.6 Option

The ASHIFT instruction supports the (– , S) option, which saturates the result.

14.5.7 Flags Affected

This instruction affects flags as follows:

- AZ is set if either result is zero; cleared if both are non-zero.
- AN is set if either result is negative; cleared if both are non-negative.
- V is set if either result overflows; cleared if neither overflows.

- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

14.5.8 Required Mode

User & Supervisor

14.5.9 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

14.5.10 Example

```
r4=r5>>>3 (v) ;                /* arithmetic right shift immediate R5.H and
                               R5.L by 3 bits (divide each half-word by 8)
                               If r5 = 0x8004 000F then the result is r4 = 0xF000 0001*/

r4=r5>>>3 (v, s) ;             // same as above, but saturate the result

r2=ashift r7 by r5.l (v) ;     /* arithmetic shift (right or left, depending on
                               sign of r5.l) R7.H and R7.L by magnitude of
                               R5.L */

r2=ashift r7 by r5.l (v, s) ;  // same as above, but saturate the result

r2=r5<<7 (v,s) ;               /* logical left shift immediate R5.H and R5.L by 7 bits,
                               saturated */
```

14.5.11 Also See

Vector Logical Shift, Arithmetic Shift (in “Shift / Rotate” Operations chapter), Logical Shift (in “Shift / Rotate” Operations chapter)

14.5.12 Special Applications

14.6 Vector Logical Shift

14.6.1 General Form

dest_reg = src_reg >> shift_magnitude (V)
dest_reg = src_reg << shift_magnitude (V)
dest_reg = LSHIFT src_reg BY shift_magnitude (V)

14.6.2 Syntax

CONSTANT SHIFT MAGNITUDE

```
Dreg = Dreg >> uimm4 (V) ;                // logical shift right, immediate (b)
Dreg = Dreg << uimm4 (V) ;                // logical shift left, immediate (b)
```

REGISTERED SHIFT MAGNITUDE

```
Dreg = LSHIFT Dreg BY Dreg_lo (V) ;       // logical shift (b)
```

14.6.3 Syntax Terminology

Dreg: R0, ..., R7
Dreg_lo: R0.L, ..., R7.L
uimm4: unsigned 4-bit field, with a range of 0 through 15

14.6.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

14.6.5 Functional Description

The Vector Logical Shift logically shifts a pair of half-word registered numbers a specified distance and direction. Though the two half-word registers are shifted at the same time, the two numbers are kept separate.

Logical shifts discard any bits shifted out of the register and backfill vacated bits with zeros.

“>>” AND “<<” SYNTAX

The two half-word registers in dest_reg are shifted by the number of places specified by shift_magnitude and the result stored into dest_reg. The data is always a pair of 16-bit half-registers. Valid shift_magnitude values are 0 – 15.

“LSHIFT” SYNTAX

Both half-word registers in src_reg are shifted by the number of places prescribed in shift_magnitude, and the result is stored into dest_reg.

For the LSHIFT versions, the sign of the shift magnitude determines the direction of the shift.

- Positive shift magnitudes produce LEFT shifts.
- Negative shift magnitudes produce RIGHT shifts.

The dest_reg and src_reg are both pairs of 16-bit half-registers.

Valid shift magnitudes for 16-bit src_reg are –16 through +15, zero included. If a number larger than these is supplied, the instruction masks and ignores the more significant bits.

This instruction does not implicitly modify the src_reg values. Optionally, dest_reg can be the same D-register as src_reg. Using the same D-register for the dest_reg and the src_reg explicitly modifies the source register at your discretion.

14.6.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if either result is zero; cleared if both are non-zero.
- AN is set if either result is negative; cleared if both are non-negative.
- V is cleared.

All other flags are unaffected.

14.6.7 Required Mode

User & Supervisor

14.6.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

14.6.9 Example

```
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 */
```

14.6.10 Also See

Vector Arithmetic Shift, Arithmetic Shift (in “Shift / Rotate” Operations chapter), Logical Shift (in “Shift / Rotate” Operations chapter)

14.6.11 Special Applications

14.7 Vector Maximum

14.7.1 General Form

dest_reg = MAX (src_reg_0, src_reg_1) (V)

14.7.2 Syntax

```
Dreg = MAX ( Dreg , Dreg ) (V) ;                // dual 16-bit operations (b)
```

14.7.3 Syntax Terminology

Dreg: R0, ..., R7

14.7.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

14.7.5 Functional 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 dest_reg.

The instruction compares the upper half-words of src_reg_0 and src_reg_1 and returns that maximum to the upper half-word of dest_reg. It also compares the lower half-words of src_reg_0 and src_reg_1 and returns that maximum to the lower half-word of dest_reg. The result is a concatenation of the two 16-bit maximum values.

The Vector Maximum instruction does not implicitly modify input values. The dest_reg can be the same D-register as one of the source registers. Doing so explicitly modifies that source register.

14.7.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if either or both result is zero; cleared if both are non-zero.
- AN is set if either or both result is negative; cleared if both are non-negative.
- V is cleared.

All other flags are unaffected.

14.7.7 Required Mode

User & Supervisor

14.7.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

14.7.9 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.

14.7.10 Also See

Vector Search, Vector Minimum, or Maximum and Minimum instructions in “Arithmetic Operations” chapter

14.7.11 Special Applications

14.8 Vector Minimum

14.8.1 General Form

dest_reg = MIN (src_reg_0, src_reg_1) (V)

14.8.2 Syntax

```
Dreg = MIN ( Dreg , Dreg ) (V) ;                    // dual 16-bit operation (b)
```

14.8.3 Syntax Terminology

Dreg: R0, ..., R7

14.8.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

14.8.5 Functional 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 dest_reg.

This instruction compares the upper half-words of src_reg_0 and src_reg_1 and returns that minimum to the upper half-word of dest_reg. It also compares the lower half-words of src_reg_0 and src_reg_1 and returns that minimum to the lower half-word of dest_reg. The result is a concatenation of the two 16-bit minimum values.

The input values are not implicitly modified by this instruction. The dest_reg can be the same D-register as one of the source registers. Doing so explicitly modifies that source register.

14.8.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if either or both result is zero; cleared if both are non-zero.
- AN is set if either or both result is negative; cleared if both are non-negative.
- V is cleared.

All other flags are unaffected.

14.8.7 Required Mode

User & Supervisor

14.8.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

14.8.9 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.

14.8.10 Also See

Vector Search, Vector Maximum, or Maximum and Minimum instructions in “Arithmetic Operations” chapter

14.8.11 Special Applications

14.9 Vector Multiply

14.9.1 Simultaneous Issue and Execution

A pair of compatible, scalar (individual) Multiply instructions from Section 10.10, “Multiply,” on page 10-24 can be combined into a single Vector Multiply instruction. The vector instruction executes the two scalar operations simultaneously and saves the results as a vector couplet.

See the Arithmetic Operations “Multiply” instruction on page 10-24 for the scalar instruction details.

Any MAC0 scalar Multiply instruction can be combined with a compatible MAC1 scalar Multiply instruction under the following conditions:

1. Both scalar instructions must share the same mode option (ex: default, IS, IU, T, etc). Exception: the MAC1 instruction can optionally employ the mixed mode (M) that does not apply to MAC0.
2. Both scalar instructions must share the same pair of source registers, but can reference different halves of those registers.
3. Both scalar operations must write to the same sized destination registers, either 16- or 32-bits.
4. The destination registers for both scalar operations must form a vector couplet, as described below.
   a. 16-bit: store results in the upper- and lower-halves of the same 32-bit Dreg. MAC0 writes to the lower half and MAC1 writes to the upper half.
   b. 32-bit: store results in valid Dreg pairs. MAC0 writes to the pair’s lower (even-numbered) Dreg and MAC1 writes to the upper (odd-numbered) Dreg.

Valid Dreg pairs are R7:6, R5:4, R3:2, and R1:0.

14.9.2 Syntax

Separate the two compatible scalar instructions with a comma to produce a vector instruction. Add a semicolon to the end of the combined instruction, as usual. The order of the MAC operations on the command line is arbitrary.

14.9.3 Instruction Length

This instruction is 32-bits long.

14.9.4 Flags Affected

This instruction affects the following flags:

- V is set if any result saturates; cleared if none saturates.
- VS is set if V is set; unaffected otherwise.

All other flags are unaffected.

14.9.5 Example

```
r2.h=r7.l*r6.h, r2.l=r7.h*r6.h ;                /* simultaneous MAC0 and MAC1 execution,
                                                16-bit results. Both results are signed
                                                fractions. */

r4.l=r1.l*r0.l, r4.h=r1.h*r0.h ;                // same as above. MAC order is arbitrary.

r0.h=r3.h*r2.l (m), r0.l=r3.l*r2.l ;            /* MAC1 multiplies a signed fraction by an
                                                unsigned fraction.  MAC0 multiplies two
                                                signed fractions. */

r5.h=r3.h*r2.h (m), r5.l=r3.l*r2.l (fu) ;       /* MAC1 multiplies signed fraction by
                                                unsigned fraction. MAC0 multiplies two
                                                unsigned fractions. */

r0.h=r3.h*r2.h, r0.l=r3.l*r2.l (is) ;           /* both MACs perform signed integer
                                                multiplication. */

r3.h=r0.h*r1.h, r3.l=r0.l*r1.l (s2rnd) ;        /* MAC1 and MAC0 multiply signed
                                                fractions. Both scale the result on the way to
                                                the destination register. */

r0.l=r7.l*r6.l, r0.h=r7.h*r6.h (iss2) ;         /* both MACs process signed integer operands
                                                and scale and round the result on the way to
                                                the destination half-registers. */

r7=r2.l*r5.l, r6=r2.h*r5.h ;                    /* both operations produce 32-bit results and
                                                save in a Dreg pair. */

r0=r4.l*r7.l, r1=r4.h*r7.h (s2rnd) ;            /* same as above, but with signed fraction
                                                scaling mode. Order of the MAC instructions
                                                makes no difference. */
```

14.10 Vector Multiply and Multiply-Accumulate

14.10.1 Simultaneous Issue and Execution

A pair of compatible, scalar (individual) instructions from...

- Section 10.11, "Multiply and Multiply-Accumulate to Accumulator," on page 10-28
- Section 10.12, "Multiply and Multiply-Accumulate to Half-Register," on page 10-31
- Section 10.13, "Multiply and Multiply-Accumulate to Data Register," on page 10-36

...can be combined into a single vector instruction. The vector instruction executes the two scalar operations simultaneously and saves the results as a vector couplet.

See the Arithmetic Operations sections listed above for the scalar instruction details.

Any MAC0 scalar instruction from the list above can be combined with a compatible MAC1 scalar instruction under the following conditions:

1. Both scalar instructions must share the same mode option (ex: default, IS, IU, T, etc). Exception: the MAC1 instruction can optionally employ the mixed mode (M) that does not apply to MAC0.
2. Both scalar instructions must share the same pair of source registers, but can reference different halves of those registers.
3. If both scalar operations write to destination Data registers, them must write to the same sized destination Data registers, either 16- or 32-bits.
4. The destination Data registers (if applicable) for both scalar operations must form a vector couplet, as described below.
   a. 16-bit: store the results in the upper- and lower-halves of the same 32-bit Dreg. MAC0 writes to the lower half, and MAC1 writes to the upper half.
   b. 32-bit: store the results in valid Dreg pairs. MAC0 writes to the pair's lower (even-numbered) Dreg, and MAC1 writes to the upper (odd-numbered) Dreg.

Valid Dreg pairs are R7:6, R5:4, R3:2, and R1:0.

14.10.2 Syntax

Separate the two compatible scalar instructions with a comma to produce a vector instruction. Add a semicolon to the end of the combined instruction, as usual. The order of the MAC operations on the command line is arbitrary.

14.10.3 Instruction Length

This instruction is 32-bits long.

14.10.4 Flags Affected

The flags reflect the results of the two scalar operations. This instruction affects flags as follows:

- V is set if any result extracted to a Dreg saturates; cleared if no Dregs saturate.
- VS is set if V is set; unaffected otherwise.
- AV0 is set if result in Accumulator A0 (MAC0 operation) saturates; cleared if A0 result does not saturate.
- AV0S is set if AV0 is set; unaffected otherwise.
- AV1 is set if result in Accumulator A1 (MAC1 operation) saturates; cleared if A1 result does not saturate.
- AV1S is set if AV1 is set; unaffected otherwise.

All other flags are unaffected.

14.10.5 Example

Result is 40-bit Accumulator

```
a1=r2.l*r3.h, a0=r2.h*r3.h ;               /* both multiply signed fractions into separate
                                           Accumulators */

a0=r1.l*r0.l, a1+=r1.h*r0.h ;              /* same as above, but sum result into A1.
                                           MAC order is arbitrary. */

a1+=r3.h*r3.l, a0=r3.h*r3.h ;              /* sum product into A1, subtract product from
                                           A0 */

a1=r3.h*r2.l (m), a0+=r3.l*r2.l ;          /* MAC1 multiplies a signed fraction in r3.h
                                           by an unsigned fraction in r2.l. MAC0
                                           multiplies two signed fractions. */

a1=r7.h*r4.h (m), a0+=r7.l*r4.l (fu) ;     /* MAC1 multiplies signed fraction by
                                           unsigned fraction. MAC0 multiplies and
                                           accumulates two unsigned fractions. */

a1+=r3.h*r2.h, a0=r3.l*r2.l (is) ;         /* both MACs perform signed integer
                                           multiplication */

a1=r6.h*r7.h, a0+=r6.l*r7.l (w32) ;        /* both MACs multiply signed fractions, sign
                                           extended, and saturate both Accumulators at
                                           bit 31 */
```

Result is 16-bit half D-register

```
r2.h=(a1=r7.l*r6.h), r2.l=(a0=r7.h*r6.h) ;           /* simultaneous MAC0 and MAC1
                                                     execution, both are signed fractions,
                                                     both products load into the Accumulators,
                                                     MAC1 into half-word registers. */
r4.l=(a0=r1.l*r0.l), r4.h=(a1+=r1.h*r0.h) ;          /* same as above, but sum result into A1. ;
                                                     MAC order is arbitrary.*/
r7.h=(a1+=r6.h*r5.l), r7.l=(a0=r6.h*r5.h) ;          // sum into A1, subtract into A0 //
r0.h=(a1=r7.h*r4.l) (m), r0.l=(a0+=r7.l*r4.l) ;      /* MAC1 multiplies a signed fraction by an
                                                     unsigned fraction. MAC0 multiplies two signed
                                                     fractions. */
r5.h=(a1=r3.h*r2.h) (m), r5.l=(a0+=r3.l*r2.l) (fu) ;/* MAC1 multiplies signed fraction by
                                                     unsigned fraction. MAC0 multiplies two
                                                     unsigned fractions. */
r0.h=(a1+=r3.h*r2.h), r0.l=(a0=r3.l*r2.l) (is) ;     /* both MACs perform signed integer
                                                     multiplication. */
r5.h=(a1=r2.h*r1.h), a0+=r2.l*r1.l ;                 /* both MACs multiply signed fractions.
                                                     MAC0 does not copy the accum result. */
r3.h=(a1=r2.h*r1.h) (m), a0=r2.l*r1.l ;              /* MAC1 multiplies signed fraction by
                                                     unsigned fraction and uses all 40 bits of A1.
                                                     MAC0 multiplies two signed fractions. */
r3.h=a1, r3.l=(a0+=r0.l*r1.l) (s2rnd) ;              /* MAC1 copies Accumulator to register half.
                                                     MAC0 multiplies signed fractions.  Both scale
                                                     the result and round on the way to the
                                                     destination register. */
r0.l=(a0+=r7.l*r6.l), r0.h=(a1+=r7.h*r6.h) (iss2) ;  /* both MACs process signed integer the way
                                                     to the destination half-registers. */
```

Result is 32-bit D-register

```
r3=(a1=r6.h*r7.h), r2=(a0=r6.l*r7.l) ;          /* simultaneous MAC0 and MAC1 execution,
                                                both are signed
                                                fractions, both products load into the
                                                Accumulators */
r4=(a0=r6.l*r7.l), r5=(a1+=r6.h*r7.h) ;         /* same as above, but sum result into A1. MAC
                                                order is arbitrary. */
r7=(a1+=r3.h*r5.h), r6=(a0-=r3.l*r5.l) ;        // sum into A1, subtract into A0
r1=(a1=r7.l*r4.l) (m), r0=(a0+=r7.h*r4.h) ;     /* MAC1 multiplies a signed fraction by an
                                                unsigned fraction. MAC0 multiplies two
                                                signed fractions. */
r5=(a1=r3.h*r7.h) (m), r4=(a0+=r3.l*r7.l) (fu);/* MAC1 multiplies signed fraction by
                                                unsigned fraction. MAC0 multiplies two
                                                unsigned fractions. */
r1=(a1+=r3.h*r2.h), r0=(a0=r3.l*r2.l) (is) ;    /* both MACs perform signed integer
                                                multiplication */
```

```
r5=(a1-=r6.h*r7.h), a0+=r6.l*r7.l ;              /* both MACs multiply signed fractions.  MAC0
                                                 does not copy the accum result */
r3=(a1=r6.h*r7.h) (m), a0-=r6.l*r7.l ;           /* MAC1 multiplies signed fraction by unsigned
                                                 fraction and uses all 40 bits of A1. MAC0
                                                 multiplies two signed fractions. */
r3=a1, r2=(a0+=r0.l*r1.l) (s2rnd) ;              /* MAC1 moves Accumulator to register.
                                                 MAC0 multiplies signed fractions.  Both scale
                                                 the result and round on the way to the
                                                 destination register. */
r0=(a0+=r7.l*r6.l), r1=(a1+=r7.h*r6.h) (iss2) ;/* both MACs process signed integer operands
                                                 and scale the result on the way to the
                                                 destination registers.*/
```

14.11 Vector Negate (Two’s Complement)

14.11.1 General Form

dest_reg = – source_reg (V)

14.11.2 Syntax

Dreg = – Dreg (V) ;	// dual 16-bit operation (b)

14.11.3 Syntax Terminology

Dreg: R0, ..., R7

14.11.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

14.11.5 Functional 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.

See Section 1.5.5, “Saturation,” on page 1-6 for a description of saturation behavior.

14.11.6 Flags Affected

This instruction affects flags as follows:

- AZ is set if either or both results are zero; cleared if both are non-zero.
- AN is set if either or both results are negative; cleared if both are non-negative.
- V is set if either or both results saturate; cleared if neither saturates.
- VS is set if V is set; unaffected otherwise.
- AC0 is set if carry occurs from either or both results; cleared if neither produces a carry.

All other flags are unaffected.

14.11.7 Required Mode

User & Supervisor

14.11.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

14.11.9 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*/
```

14.11.10 Also See

Negate (Two’s Complement) in the “Arithmetic Operations” chapter.

14.11.11 Special Applications

14.12 Vector Pack

14.12.1 General Form

Dest_reg = PACK (src_half_0, src_half_1)

14.12.2 Syntax

Dreg = PACK (Dreg_lo_hi , Dreg_lo_hi) ; // (b)

14.12.3 Syntax Terminology

Dreg: R0, ..., R7

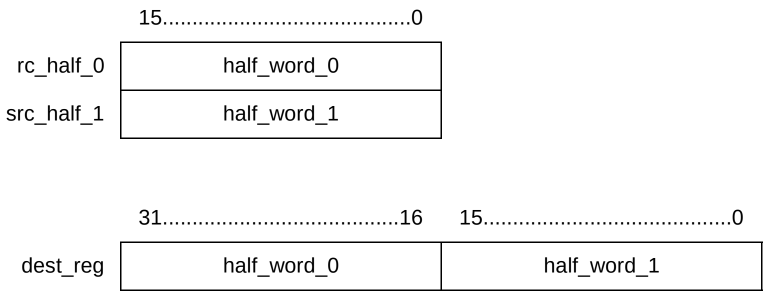
Dreg_lo_hi: R0.L, ..., R7.L, R0.H, ..., R7.H

14.12.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

14.12.5 Functional Description

The Vector Pack instruction packs two 16-bit half-word numbers into the halves of a 32-bit data register.



14.12.6 Flags Affected

None.

14.12.7 Required Mode

User & Supervisor

14.12.8 Parallel Issue

The 32-bit versions of this instruction can be issued in parallel with specific other 16-bit instructions. For details, see Chapter 15, “Issuing Parallel Instructions.”

14.12.9 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
```

14.12.10 Also See

Quad 8-Bit Pack (in “Video Pixel Operations” chapter)

14.12.11 Special Applications

```
        // If r4.l = 0xDEAD and r5.l = 0xBEEF, then . . .
r3 = pack (r4.l, r5.l) ;
        // . . . produces r3 = 0xDEAD BEEF
```

14.13 Vector Search

14.13.1 General Form

(dest_pointer_hi, dest_pointer_lo) = SEARCH src_reg (searchmode)

14.13.2 Syntax

```
( Dreg , Dreg ) = SEARCH Dreg (searchmode) ;          // (b)
```

14.13.3 Syntax Terminology

Dreg: R0, ..., R7

searchmode: (GT), (GE), (LE), or (LT)

14.13.4 Instruction Length

In the syntax, comment (b) identifies 32-bit instruction length.

14.13.5 Functional 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 “Example” 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.

14.13.6 Modes

The four supported compare modes are specified by the mandatory searchmode flag, that can be any one of the following

Table 14-2. Compare Modes

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.

14.13.7 Flags Affected

None.

14.13.8 Required Mode

User & Supervisor

14.13.9 Parallel Issue

This instruction can be issued in parallel with one 32-bit instruction, provided that load is based on the P0 pointer register. This condition is the only case that supports parallel issue with Vector Search.

14.13.10 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

/* Initilaize R2. */
        r2=[p0++];

        lsetup (search_loop, search_loop) lc0=p1>>1 ;       // set up the loop
search_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 to find the
single, last minimum in the array. Then choose the corresponding
pointer from R1 or R0.

Note: In this example, the resulting pointers are 4 past the actual surviving array element due to the
post-increment operation. Subtract 4 before using the final pointer value. */
```

14.13.11 Also See

Vector Maximum, Vector Minimum, or Maximum and Minimum instructions explained in Chapter 15, “Issuing Parallel Instructions.”

14.13.12 Special Applications

This instruction is used in a loop to locate an element in a vector according to the element’s value.

ISSUING PARALLEL INSTRUCTIONS 15

15.6.2 One Ireg and One Memory Access Instruction in Parallel 15-7

15.1 Summary

The Blackfin 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 instruction 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.

15.2 Supported Parallel Combinations

The diagram below illustrates the combinations for parallel issue that Blackfin supports.

32-bit ALU/MAC instruction	16-bit Instruction	16-bit Instruction

15.3 Parallel Issue Syntax

The syntax of a parallel issue instruction is…

32-bit ALU/MAC instruction || *A 16-bit instructions* || *A 16-bit instruction*;

The vertical bar (||) indicates that the following instruction is to be issued in parallel with the previous instruction. Note that 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 no-op automatically inserted into the unused 16-bit slot.

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 instructions* || *A 16-bit instruction*;

See the MNOP (32-bit NOP) instruction description in Section 11.10, “No Op,” on page 11-20 of this document. 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.

15.4 32-Bit ALU/MAC Instructions

The list of 32-bit instructions that can be in a parallel instruction are shown below.

Table 15-1. 32-Bit DSP Instructions

SECTION	INSTRUCTION NAME	NOTES
ARITHMETIC OPERATIONS	Absolute Value	
	Add	Only the versions that support optional saturation.
	Exponent Detection	
	Maximum	
	Minimum	
	Modify – Decrement (for Accumulators, only)	
	Modify – Increment (for Accumulators, only)	Accumulator versions, only.
	Negate (Two’s Complement)	Accumulator versions, only.
	Round Half-Word	
	Round – 12 Bit	
	Round – 20 Bit	
	Saturate	
	Sign Bit	
	Subtract	Saturating versions, only.
BIT OPERATIONS	Bit Field Deposit	
	Bit Field Extract	
	Bit Multiplex	
	Ones Population Count	
LOGICAL OPERATIONS	Bit-Wise XOR	
MOVE	Move Register	40-bit Accumulator versions, only.
	Move Register Half	
SHIFT / ROTATE OPERATIONS	Arithmetic Shift	Saturating and Accumulator versions, only.
	Logical Shift	32-bit instruction size versions, only.
	Rotate	
EXTERNAL EVENT MANAGEMENT	No Op	32-bit MNOP, only
VECTOR OPERATIONS	Compare-Select (VIT_MAX)	
	Add on Sign	
	Multiply and Multiply-Accumulate to Accumulator	
	Multiply and Multiply-Accumulate to Half-Register	
	Multiply and Multiply-Accumulate to Data Register	

Table 15-1. 32-Bit DSP Instructions

SECTION	INSTRUCTION NAME	NOTES
	Vector Absolute Value	
	Vector Add / Subtract	
	Vector Arithmetic Shift	
	Vector Logical Shift	
	Vector Maximum	
	Vector Minimum	
	Multiply	
	Vector Negate (Two's Complement)	
	Vector Pack	
	Vector Search	
VIDEO PIXEL OPERATIONS	Byte Align	
	Disable Alignment Exception for Load	
	Quad 8-Bit Subtract-Absolute-Accumulate	
	Quad 8-Bit SAA Accumulator Extract	
	Quad 8-Bit Add	
	Quad 8-Bit Subtract	
	Quad 8-Bit Average – Byte	
	Quad 8-Bit Average – Half-Word	
	Quad 8-Bit Add / Clip	
	Quad 8-Bit Pack	
	Quad 8-Bit Unpack	

15.5 16-Bit Instructions

The two 16-bit instructions in a multi-issue instruction must each be from Group1 and Group2 instructions shown below.

Table 15-2. Group1 Compatible 16-Bit Instructions

SECTION	INSTRUCTION NAME	NOTES
ARITHMETIC OPERATIONS	Add Immediate	Ireg versions only.
	Modify – Decrement	Ireg versions only.
	Modify – Increment	Ireg versions only.
	Subtract Immediate	Ireg versions only.
LOAD / STORE	Load Pointer Register	
	Load Data Register	
	Load Half-Word – Zero-Extended	
	Load Half-Word – Sign-Extended	
	Load High Data Register Half	
	Load Low Data Register Half	
	Load Byte – Zero-Extended	
	Load Byte – Sign-Extended	
	Store Pointer Register	
	Store Data Register	
	Store High Data Register Half	
	Store Low Data Register Half	
	Store Byte	

Table 15-3. Group2 Compatible 16-Bit Instructions

SECTION	INSTRUCTION NAME	NOTES
LOAD / STORE	Load Data Register	Ireg versions, only
	Load High Data Register Half	Ireg versions, only
	Load Low Data Register Half	Ireg versions, only
	Store Data Register	Ireg versions, only
	Store High Data Register Half	Ireg versions, only
	Store Low Data Register Half	Ireg versions, only

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.
- If the two 16-bit instructions are memory access instructions, then both cannot use P-registers as address registers. In this case, at least one memory access instruction must be an I-register version.

15.6 Examples

15.6.1 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++] ;
```

15.6.2 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++];
```

15.6.3 One Ireg Instruction in Parallel

```
// Multiply-Accumulate to a Data register while incrementing an Ireg.
    r6=(a0+=r3.h*r2.h)(fu)  ||  i2-=m0;
```

INDEX