ADSP-2136x SHARC® Processor Programming Reference

Revision 1.1, March 2007

Part Number
82-000500-01

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

ANALOG DEVICES

Contents

PROCESSING ELEMENTS

REGISTERS

PREFACE

Thank you for purchasing and developing systems using the ADSP-2136x SHARC® processor from Analog Devices.

Purpose of This Manual

The *ADSP-2136x SHARC Processor Programming Reference* provides architectural and programming information about the ADSP-2136x SHARC processor. The architectural descriptions cover the processor's functional blocks and buses, including features and processes that they support. The programming information covers the Instruction Set and Compute operations. The companions to this manual are the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* and the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*. These manuals provide information on the I/O capabilities and peripherals supported on these processors. For timing, electrical, and package specifications, see the processor specific data sheet listed in "Related Documents" on page xxix.

Intended Audience

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

Manual Contents

This manual provides detailed information about the ADSP-2136x processor family in the following chapters:

- Chapter 1, "Introduction"
  Provides an architectural overview of the ADSP-2136x processors.
- Chapter 2, "Processing Elements"
  Describes the arithmetic/logic units (ALUs), multiplier/accumulator units, and shifter. The chapter also discusses data formats, data types, and register files.
- Chapter 3, "Program Sequencer"
  Describes the operation of the program sequencer, which controls program flow by providing the address of the next instruction to be executed. The chapter also discusses loops, subroutines, jumps, interrupts, exceptions, and the `IDLE` instruction.
- Chapter 4, "Data Address Generators"
  Describes the Data Address Generators (DAGs), addressing modes, how to modify DAG and pointer registers, memory address alignment, and DAG instructions.
- Chapter 5, "Memory"
  Describes aspects of processor memory including internal memory, address and data bus structure, and memory accesses.

- Chapter 6, "JTAG Test Emulation Port"
  Discusses the JTAG standard and how to use the ADSP-2136x processors in a test environment. Includes boundary-scan architecture, instruction and boundary registers, and breakpoint control registers.

- Chapter 7 "Timer"
  Describes the three general purpose timers that can be configured in any of three modes: pulse width modulation, pulse width count and capture, and external event watchdog modes.

- Chapter 8, "Instruction Set"
  Provides reference information for the machine language opcode for the processor.

- Chapter 9, "Computations Reference"
  Describes each compute operation in detail, including its assembly language syntax and opcode field. Compute operations execute in the multiplier, the ALU, and the shifter.

- Appendix A, "Instruction Set Quick Reference"
  The instruction set summary provides a syntax summary for each instruction and includes a cross reference to each instruction's reference page.

- Appendix B, "Registers"
  Provides register and bit descriptions for all of the registers that are used to control the operation of the ADSP-2136x processor core.

This programming reference is a companion document to the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* and the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors.*

What's New in This Manual

This is revision 1.1 of the *ADSP-2136x SHARC Processor Programming Reference*. The only changes for this revisions are corrections to cross references (and links in the online version of the book).

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at `http://www.analog.com/processors/technicalSupport`
- E-mail tools questions to `processor.tools.support@analog.com`
- E-mail processor questions to
  `processor.support@analog.com (World wide support)`
  `processor.europe@analog.com (Europe support)`
  `processor.china@analog.com (China support)`
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:

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

Supported Processors

The following is the list of Analog Devices, Inc. processors supported in VisualDSP++®.

TigerSHARC® (ADSP-TSxxx) Processors

The name *TigerSHARC* refers to a family of floating-point and fixed-point [8-bit, 16-bit, and 32-bit] processors. VisualDSP++ currently supports the following TigerSHARC families: ADSP-TS101 and ADSP-TS20x.

SHARC (ADSP-21xxx) Processors

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

Blackfin® (ADSP-BFxxx) Processors

The name *Blackfin* refers to a family of 16-bit, embedded processors. VisualDSP++ currently supports the following Blackfin families: ADSP-BF53x and ADSP-BF56x.

Product Information

You can obtain product information from the Analog Devices Web site, from the product CD-ROM, or from the printed publications (manuals).

Analog Devices is online at `www.analog.com`. Our Web site provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

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

Registration

Visit `www.myanalog.com` to sign up. Click **Register** to use `MyAnalog.com`. Registration takes about five minutes and serves as a means to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your e-mail address.

Processor Product Information

For information on embedded processors and DSPs, visit our Web site at `www.analog.com/processors`, which provides access to technical publications, data sheets, application notes, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- E-mail questions or requests for information to
  `processor.support@analog.com (World wide support)`
  `processor.europe@analog.com (Europe support)`
  `processor.china@analog.com (China support)`

- Fax questions or requests for information to
  1-781-461-3010 (North America)
  +49-89-76903-157 (Europe)
- Access the FTP Web site at
  `ftp ftp.analog.com` (or `ftp 137.71.25.69)`
  `ftp://ftp.analog.com`

Related Documents

The following publications that describe the ADSP-2136x processors can be ordered from any Analog Devices sales office:

- *ADSP-21362 SHARC Processor Data Sheet*
- *ADSP-21363 SHARC Processor Data Sheet*
- *ADSP-21364 SHARC Processor Data Sheet*
- *ADSP-21365 SHARC Processor Data Sheet*
- *ADSP-21366 SHARC Processor Data Sheet*
- *ADSP-21367 SHARC Processor Preliminary Data Sheet*
- *ADSP-21368 SHARC Processor Preliminary Data Sheet*
- *ADSP-21369 SHARC Processor Preliminary Data Sheet*
- *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors*
- *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*

For information on product related development software and Analog Devices processors, see these publications:

- *VisualDSP++ User's Guide*
- *VisualDSP++ C/C++ Compiler and Library Manual*
- *VisualDSP++ Assembler and Preprocessor Manual*
- *VisualDSP++ Linker and Utilities Manual*
- *VisualDSP++ Kernel (VDK) User's Guide*

Visit the Technical Library Web site to access all processor and tools manuals and data sheets:

`http://www.analog.com/processors/resources/technicalLibrary`

Online Technical Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, the Dinkum Abridged C++ library, and Flexible License Manager (FlexLM) network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest. For easy printing, supplementary `.PDF` files of most manuals are also provided.

Each documentation file type is described as follows.

File	Description
`.CHM`	Help system files and manuals in Help format
`.HTM` or `.HTML`	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the `.HTML` files requires a browser, such as Internet Explorer 4.0 (or higher).
`.PDF`	VisualDSP++ and processor manuals in Portable Documentation Format (PDF). Viewing and printing the `.PDF` files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

If documentation is not installed on your system as part of the software installation, you can add it from the VisualDSP++ CD-ROM at any time by running the Tools installation. Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or the Analog Devices Web site.

Accessing Documentation From VisualDSP++

From the VisualDSP++ environment:

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.
- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

Accessing Documentation From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (`.CHM`) are located in the `Help` folder, and `.PDF` files are located in the `Docs` folder of your VisualDSP++ installation CD-ROM. The `Docs` folder also contains the Dinkum Abridged C++ library and the FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click the `vdsp-help.chm` file, which is the master Help system, to access all the other `.CHM` files.
- Double-click any file that is part of the VisualDSP++ documentation set.

Using the Windows Start Button

- Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, and **VisualDSP++ Documentation**.
- Access the `.PDF` files by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, **Documentation for Printing**, and the name of the book.

Accessing Documentation From the Web

Download manuals at the following Web site:
`http://www.analog.com/processors/technical_library`

Select a processor family and book title. Download archive (`.ZIP`) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**) and follow the prompts.

VisualDSP++ Documentation Set

To purchase VisualDSP++ manuals, call **1-603-883-2430**. The manuals may be purchased only as a kit.

If you do not have an account with Analog Devices, you are referred to Analog Devices distributors. For information on our distributors, log onto `http://www.analog.com/salesdir`.

Hardware Tools Manuals

To purchase EZ-KIT Lite® and In-Circuit Emulator (ICE) manuals, call **1-603-883-2430**. The manuals may be ordered by title or by product number located on the back cover of each manual.

Processor Manuals

Hardware reference and instruction set reference manuals may be ordered through the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**), or downloaded from the Analog Devices Web site. Manuals may be ordered by title or by product number located on the back cover of each manual.

Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. Only production (final) data sheets (Rev. 0, A, B, C, and so on) can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**); they also can be downloaded from the Web site.

To have a data sheet faxed to you, call the Analog Devices Faxback System at **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. If the data sheet you want is not listed, check for it on the Web site.

Conventions

Text conventions used in this manual are identified and described as follows.

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

ⓘ Additional conventions, which apply only to specific chapters, may appear throughout this document.

1 INTRODUCTION

The ADSP-2136x processors are high performance 32-bit processors used for medical imaging, communications, military, audio, test equipment, 3D graphics, speech recognition, motor control, imaging, and other applications. By adding on-chip SRAM, integrated I/O peripherals, and an additional processing element for single-instruction, multiple-data (SIMD) support, this processor builds on the ADSP-21000 family processor core to form a complete system-on-a-chip.

The ADSP-2136x processors are comprised of two distinct groups, the ADSP-21362/3/4/5/6 processors (see Figure 1-1 on page 1-3 and Table 1-1 on page 1-11), and the ADSP-21367/8/9 processors (see Figure 1-2 on page 1-4 and Table 1-2 on page 1-12). The groups are differentiated by, on-chip memories, peripheral choices, packaging, and operating speeds. However, the core processor operates in the same way in both groups so this manual applies to both groups. Where differences exist (such as external memory interfacing) they will be noted.

For specific information on the peripherals associated with each group, two manuals are available: the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* and the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors.*

ADSP-2136x Design Advantages

A digital signal processor's data format determines its ability to handle signals of differing precision, dynamic range, and signal-to-noise ratios. Because floating-point math reduces the need for scaling and probability of overflow, using a floating-point processor can ease algorithm and software development. The extent to which this is true depends on the floating-point processor's architecture. Consistency with IEEE workstation simulations and the elimination of scaling are clearly two ease-of-use advantages. High level language programmability, large address spaces, and wide dynamic range allow system development time to be spent on algorithms and signal processing concerns, rather than assembly language coding, code paging, and error handling. The ADSP-2136x processors are highly integrated, 32-bit floating-point processors that provide many of these design advantages.

The SHARC processor architecture balances a high performance processor core with high performance program memory (PM), data memory (DM), and input/output (I/O) buses. In the core, every instruction can execute in a single cycle. The buses and instruction cache provide rapid, unimpeded data flow to the core to maintain the execution rate.

Figure 1-1 shows a detailed block diagram of the processor, illustrating the following architectural features:

- Two processing elements (PEx and PEy), each containing 32-bit IEEE floating-point computation units—multiplier, arithmetic logic unit (ALU), shifter, and data register file
- Program sequencer with related instruction cache, interval timer, and data address generators (DAG1 and DAG2)
- Up to 3M bit on-chip SRAM
- IOP with integrated direct memory access (DMA) controller, serial peripheral interface (SPI) compatible port, and serial ports (SPORTs) for point-to-point multiprocessor communications.
- JTAG test access port for emulation

- External port for interfacing to off-chip SDRAM (ADSP-21367/8/9 processors) and configuring a shared memory system with up to four other ADSP-21368 SHARC processors
- Parallel port for interfacing to off-chip memory and peripherals (ADSP-21362/3/4/5/6 processors)

Figure 1-1 also shows the three on-chip buses of the ADSP-2136x processors: the PM bus, DM bus, and I/O bus. The PM bus provides access to either instructions or data. During a single cycle, these buses let the processor access two data operands from memory, access an instruction (from the cache), and perform a DMA transfer.

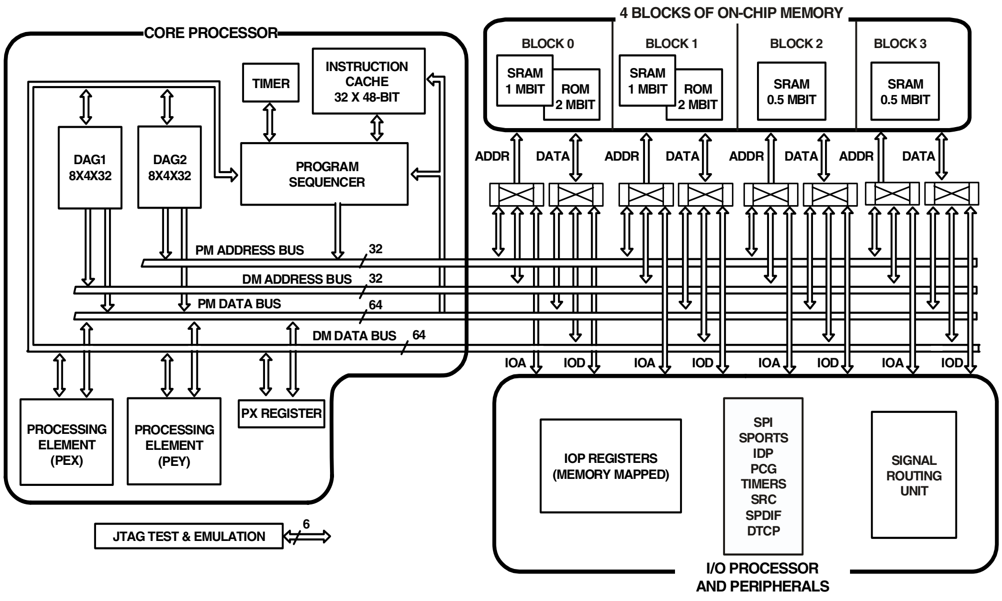


Figure 1-1. ADSP-21362/3/4/5/6 SHARC Processor Block Diagram

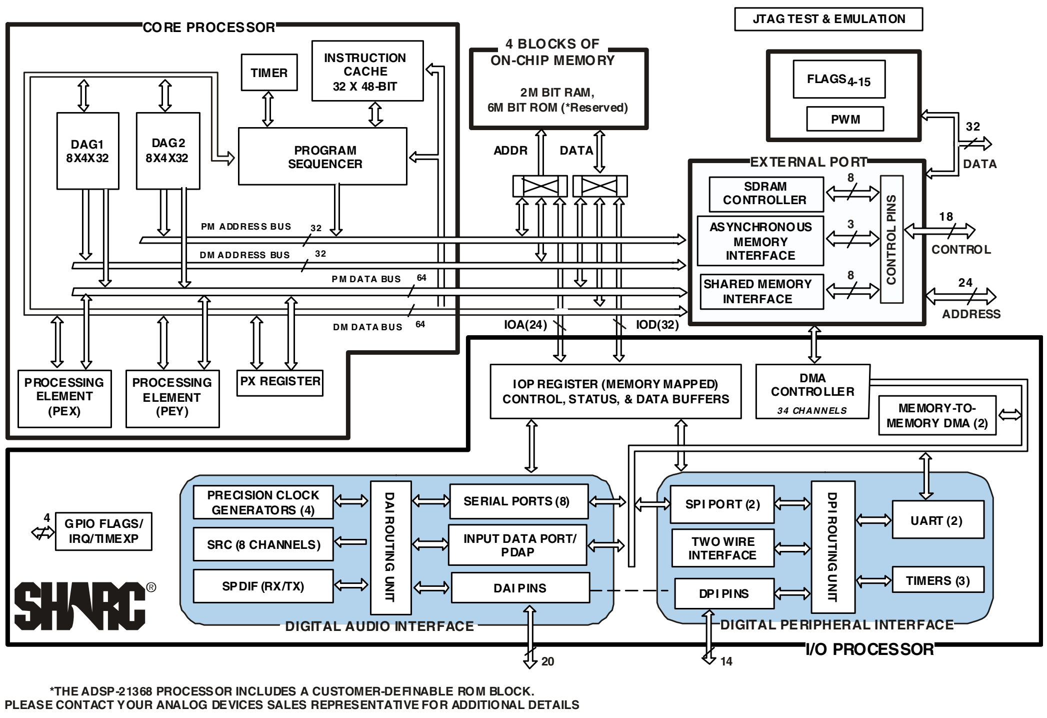


Figure 1-2. ADSP-21367/8/9 SHARC Processor Block Diagram

The ADSP-2136x processors address the five central requirements for signal processing:

1. **Fast, flexible arithmetic.** The ADSP-21000 family processors execute all instructions in a single cycle. They provide fast cycle times and a complete set of arithmetic operations. The processor is IEEE floating-point compatible and allows either interrupt on arithmetic exception or latched status exception handling.

2. **Unconstrained data flow.** The ADSP-2136x processors have a Super Harvard Architecture combined with a ten-port data register file. For more information, see "Data Register File" on page 2-37. In every cycle, the processor can write or read two operands to or

from the register file, supply two operands to the ALU, supply two operands to the multiplier, and receive three results from the ALU and multiplier. The processor's 48-bit orthogonal instruction word supports parallel data transfers and arithmetic operations in the same instruction.

3. **40-Bit extended precision.** The processor handles 32-bit IEEE floating-point format, 32-bit integer and fractional formats (twos-complement and unsigned), and extended-precision 40-bit floating-point format. The processors carry extended precision throughout their computation units, limiting intermediate data truncation errors (up to 80 bits of precision are maintained during multiply-accumulate operations).
4. **Dual address generators.** The processor has two data address generators (DAGs) that provide immediate or indirect (pre- and post-modify) addressing. Modulus, bit-reverse, and broadcast operations are supported with no constraints on data buffer placement.
5. **Efficient program sequencing.** In addition to zero-overhead loops, the processor supports single-cycle setup and exit for loops. Loops are both nestable (six levels in hardware) and interruptable. The processors support both delayed and non-delayed branches.

ADSP-2136x Architectural Overview

The ADSP-2136x processors form a complete system-on-a-chip, integrating a large, high speed SRAM and I/O peripherals supported by a dedicated I/O bus. The following sections summarize the features of each functional block in the ADSP-2136x architecture, which appears in Figure 1-1.

Processor Core

The processor core consists of two processing elements (each with three computation units and data register file), a program sequencer, two DAGs, a timer, and an instruction cache. All processing occurs in the processor core.

Processing Elements

The processor core contains two processing elements: PEx and PEy. Each element contains a data register file and three independent computation units: an arithmetic logic unit (ALU), a multiplier with an 80-bit fixed-point accumulator, and a shifter. For meeting a wide variety of processing needs, the computation units process data in three formats: 32-bit fixed-point, 32-bit floating-point, and 40-bit floating-point. The floating-point operations are single-precision IEEE-compatible. The 32-bit floating-point format is the standard IEEE format, whereas the 40-bit extended-precision format has eight additional least significant bits (LSBs) of mantissa for greater accuracy.

The ALU performs a set of arithmetic and logic operations on both fixed-point and floating-point formats. The multiplier performs floating-point or fixed-point multiplication and fixed-point multiply/accumulate or multiply/cumulative-subtract operations. The shifter performs logical and arithmetic shifts, bit manipulation, bit-wise field deposit and extraction, and exponent derivation operations on 32-bit operands. These computation units complete all operations in a single cycle; there is no computation pipeline. The output of any unit may serve as the input of any unit on the next cycle. All units are connected in parallel, rather than serially. In a multifunction computation, the ALU and multiplier perform independent, simultaneous operations.

Each processing element has a general-purpose data register file that transfers data between the computation units and the data buses and stores intermediate results. A register file has two sets (primary and secondary) of

16 general-purpose registers each for fast context switching. All of the registers are 40 bits wide. The register file, combined with the core processor's Super Harvard Architecture, allows unconstrained data flow between computation units and internal memory.

Primary processing element (PEx). PEx processes all computational instructions whether the processor is in single-instruction, single-data (SISD) or single-instruction, multiple-data (SIMD) mode. This element corresponds to the computational units and register file in previous ADSP-21000 family processors.

Secondary processing element (PEy). PEy processes each computational instruction in lock-step with PEx, but only processes these instructions when the processor is in SIMD mode. Because many operations are influenced by this mode, more information on SIMD is available in multiple locations:

- For information on PEy operations, see "Processing Elements" on page 2-1.
- For information on data addressing in SIMD mode, see "Addressing in SISD and SIMD Modes" on page 4-20.
- For information on data accesses in SIMD mode, see "SISD, SIMD, and Broadcast Load Modes" on page 5-37.
- For information on SIMD programming, see "Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference.

Program Sequence Control

Internal controls for program execution come from four functional blocks: program sequencer, data address generators, core timer, and instruction cache. Two dedicated address generators and a program sequencer supply addresses for memory accesses. Together the sequencer and data address generators allow computational operations to execute with maximum efficiency since the computation units can be devoted exclusively to processing data. With its instruction cache, the ADSP-2136x processors can simultaneously fetch an instruction from the cache and access two data operands from memory. The DAGs also provide built-in support for zero-overhead circular buffering.

Program sequencer. The program sequencer supplies instruction addresses to program memory. It controls loop iterations and evaluates conditional instructions. With an internal loop counter and loop stack, the processors execute looped code with zero overhead. No explicit jump instructions are required to loop or to decrement and test the counter. To achieve a high execution rate while maintaining a simple programming model, the processor employs a five stage pipeline to process instructions — fetch1, fetch2, decode, address and execute. **For more information, see "Instruction Pipeline" on page 3-2.**

Data address generators. The DAGs provide memory addresses when data is transferred between memory and registers. Dual data address generators enable the processor to output simultaneous addresses for two operand reads or writes. DAG1 supplies 32-bit addresses for accesses using the DM bus. DAG2 supplies 32-bit addresses for memory accesses over the PM bus.

Each DAG keeps track of up to eight address pointers, eight address modifiers, and for circular buffering eight base-address registers and eight buffer-length registers. A pointer used for indirect addressing can be modified by a value in a specified register, either before (pre-modify) or after (post-modify) the access. A length value may be associated with each pointer to perform automatic modulo addressing for circular data buffers. The circular buffers can be located at arbitrary boundaries in memory. Each DAG register has a secondary register that can be activated for fast context switching.

Circular buffers allow efficient implementation of delay lines and other data structures required in digital signal processing They are also commonly used in digital filters and Fourier transforms. The DAGs automatically handle address pointer wraparound, reducing overhead, increasing performance, and simplifying implementation.

Interrupts. The ADSP-2136x processors have three external hardware interrupts. The processor also provides three general-purpose interrupts, and a special interrupt for reset. The processor has internally-generated interrupts for the timer, DMA controller operations, circular buffer overflow, stack overflows, arithmetic exceptions, and user-defined software interrupts.

For the general-purpose interrupts and the internal timer interrupt, the processor automatically stacks the arithmetic status (`ASTATx`) register and mode (`MODE1`) registers in parallel with the interrupt servicing, allowing 15 nesting levels of very fast service for these interrupts.

Context switch. Many of the processor's registers have secondary registers that can be activated during interrupt servicing for a fast context switch. The data registers in the register file, the DAG registers, and the multiplier result register all have secondary registers. The primary registers are active at reset, while the secondary registers are activated by control bits in a mode control register.

Timer. The core's programmable interval timer provides periodic interrupt generation. When enabled, the timer decrements a 32-bit count register every cycle. When this count register reaches zero, the ADSP-2136x processors generate an interrupt and asserts their timer expired output. The count register is automatically reloaded from a 32-bit period register and the countdown resumes immediately.

Instruction cache. The program sequencer includes a 32-word instruction cache that effectively provides three-bus operation for fetching an instruction and two data values. The cache is selective; only instructions whose fetches conflict with data accesses using the PM bus are cached. This caching allows full speed execution of core, looped operations such as digital filter multiply-accumulates, and FFT butterfly processing. For more information on the cache, refer to "Using the Cache" on page 3-8.

Processor Internal Buses

The processor core has six buses: PM address, PM data, DM address, DM data, I/O address, and I/O data. The PM bus is used to fetch instructions from memory, but may also be used to fetch data. The DM bus can only be used to fetch data from memory. The I/O bus is used solely by the IOP to facilitate DMA transfers. In conjunction with the cache, this Super Harvard Architecture allows the core to fetch an instruction and two pieces of data in the same cycle that a data word is moved between memory and a peripheral. This architecture allows dual data fetches, when the instruction is supplied by the cache.

Bus capacities. The PM and DM address buses are both 32 bits wide, while the PM and DM data buses are both 64 bits wide.

These two buses provide a path for the contents of any register in the processor to be transferred to any other register or to any data memory location in a single cycle. When fetching data over the PM or DM bus, the address comes from one of two sources: an absolute value specified in the instruction (direct addressing) or the output of a data address generator (indirect addressing). These two buses share the same port of the memory.

Each memory block also has a dedicated I/O address bus and I/O data bus to let the I/O processor access internal memory for DMA without delaying the processor core (in the absence of memory block conflict). The I/O address bus is 18 bits wide, and the I/O data bus is 32 bits wide.

Data transfers. Nearly every register in the processor core is classified as a universal register (*Ureg*). Instructions allow the transfer of data between any two universal registers or between a universal register and memory. This support includes transfers between control registers, status registers, and data registers in the register file. The PM bus connect (PX) registers permit data to be passed between the 64-bit PM data bus and the 64-bit DM data bus, or between the 40-bit register file and the PM data bus. These registers contain hardware to handle the data width difference. For more information, see "Processing Element Registers" on page B-22.

Processor Peripherals

The term processor peripherals refers to the multiple on-chip functional blocks used to communicate with off-chip devices. The ADSP-21362/3/4/5/6 peripherals include the JTAG, parallel, serial, SPI ports, DAI components (PCG, timers, and IDP), and any external devices that connect to the processor. The ADSP-21367/8/9 processors peripherals include the JTAG, external, serial, DAI components (PCG, Timers, and IDP), DPI components (two UARTs, two SPIs, three timers, and a two wire interface port) and any external devices that connect to the processor. For complete information on using peripherals, see the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* or the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

Table 1-1 and Table 1-2 provide details on the various options available from each processor group.

Table 1-1. ADSP-21362/3/4/5/6 SHARC Processor Features

Feature	ADSP-21362	ADSP-21363	ADSP-21364	ADSP-21365[1]	ADSP-21366
RAM	3M bit	3M bit	3M bit	3M bit	3M bit
ROM	4M bit	4M bit	4M bit	4M bit	4M bit
Audio Decoders in ROM[2]	No	No	No	Yes	Yes
Pulse Width Modulation	Yes	Yes	Yes	Yes	Yes
S/PDIF	Yes	No	Yes	Yes	Yes

Table 1-1. ADSP-21362/3/4/5/6 SHARC Processor Features (Cont'd)

Feature	ADSP-21362	ADSP-21363	ADSP-21364	ADSP-21365[1]	ADSP-21366
SRC Performance	128db	No SRC	140dB	128dB	128dB
Package Option[3]	136 Ball BGA 144 Lead LQFP	136 Ball BGA 144 Lead LQFP	136 Ball BGA 144 Lead LQFP	136 Ball BGA 144 Lead LQFP	136 Ball BGA 144 Lead LQFP
Processor Speed	333 MHz	333 MHz	333 MHz	333 MHz	333 MHz

1 The ADSP-21365 provides the Digital Transmission Content Protection protocol, a proprietary security protocol. Contact your Analog Devices sales office for more information.

2 Audio decoding algorithms include PCM, Dolby Digital EX, Dolby Prologic IIx, DTS 96/24, Neo:6, DTS ES, MPEG2 AAC, MP3, and functions like bass management, delay, speaker equalization, graphic equalization, and more. Decoder/post-processor algorithm combination support vary, depending upon the chip version and the system configurations. Please visit www.analog.com/SHARC for complete information.

3 Analog Devices offers these packages in lead (Pb) free versions.

Table 1-2. ADSP-21367/8/9 SHARC Processor Features

Feature	ADSP-21367	ADSP-21368	ADSP-21369
RAM	2M bit	2M bit	2M bit
ROM	6M bit	6M bit[1]	6M bit[1]
Audio Decoders in ROM[2]	Yes	No	No
Pulse Width Modulation	Yes	Yes	Yes
S/PDIF	Yes	Yes	Yes
Shared Memory	No	Yes	No
SRC Performance	128dB	140dB	128dB
Package Option[3]	256 Ball SBGA 208 Lead MQFP	256 Ball BGA	256 Ball BGA 208 Lead MQFP
Processor Speed	400 MHz	400 MHz	400 MHz

1 The ADSP-21368/21369 processors includes a customer-definable ROM block. Please contact your Analog Devices sales representative for additional details.

2 Audio decoding algorithms include PCM, Dolby Digital EX, PCM, Dolby Digital EX, Dolby Prologic IIx, DTS 96/24, Neo:6, DTS ES, MPEG2 AAC, MPEG2 2channel, MP3, and functions like bass management, delay, speaker equalization, graphic equalization, and more. Decoder/post-processor algorithm combination support vary depending upon the chip version and the system configurations. Please visit www.analog.com/SHARC for complete information.

3 Analog Devices offers these packages in lead (Pb) free versions.

Internal Memory (SRAM)

The individual ADSP-2136x products contain varying amounts of memory. For example, the ADSP-21362/3/4/5/6 processors provide 3M bits of internal SRAM and 4M bits of internal ROM, which is organized into four separate blocks. The memory and separate on-chip buses allow two data transfers from the core and one from I/O, all in a single cycle.

All of the memory can be accessed as 16-, 32-, 48-, or 64-bit words. On the ADSP-2136x processors, the memory can be configured as a maximum of 96K words of 32-bit data, 192K words of 16-bit data, 64K words of 48-bit instructions (and 40-bit data), or combinations of different word sizes up to 3.0M bit. For specific memory configurations, see the product model specific data sheet.

The processor also supports a 16-bit floating-point storage format, which effectively doubles the amount of data that may be stored on chip. Conversion between the 32-bit floating-point and 16-bit floating-point formats completes in a single instruction.

While each memory block can store combinations of code and data, accesses are most efficient when one block stores data (using the DM bus for transfers) and the other block stores instructions and data (using the PM bus for transfers). Using the DM and PM buses in this way (with one dedicated to each memory block) assures single-cycle execution with two data transfers. In this case, the instruction must be available in the cache. The processor also maintains single-cycle execution when one of the data operands is transferred to or from off chip, using the processor's parallel port.

Timers

In addition to the core's programmable interval timer, the ADSP-2136x processors have three programmable interval timers that generate periodic interrupts. Each timer can be independently set to operate in one of three modes:

- Pulse waveform generation mode
- Pulse width count/capture mode
- External event watchdog mode

Each timer has one bidirectional pin and four registers that implement its mode of operation. These registers are a 7-bit configuration register, a 32-bit count register, a 32-bit period register, and a 32-bit pulse width register. A single status register supports all three timers. A bit in each timer's configuration register enables or disables the corresponding timer independently of the others.

JTAG Port

The JTAG port supports the IEEE standard 1149.1 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system. Emulators use the JTAG port to monitor and control the processor during emulation. Emulators using this port provide full speed emulation with access to inspect and modify memory, registers, and processor stacks. JTAG-based emulation is non-intrusive and does not effect target system loading or timing.

Rom Based Security

For those devices with application code in the on-chip ROM, an optional ROM security feature is included. This feature provides hardware support for securing user software code by preventing unauthorized reading from the enabled code. The processor does not boot-load any external code, executing exclusively from internal ROM. The processor also is not freely accessible via the JTAG port. Instead a 64-bit key is assigned to the user. This key must be scanned in through the JTAG or Test Access Port. The device ignores a wrong key. Emulation features and external boot modes are only available after the correct key is scanned.

Development Tools

The ADSP-2136x SHARC processors are supported by VisualDSP++, an easy to use Integrated Development and Debugging Environment (IDDE). VisualDSP++ allows you to manage projects from start to finish from within a single, integrated interface. Because the project development and debug environments are integrated, you can move easily between editing, building, and debugging activities.

Differences From Previous SHARC Processors

This section identifies differences between the ADSP-2136x processors and previous SHARC processors: ADSP-21161, ADSP-21160, ADSP-21060, ADSP-21061, ADSP-21062, and ADSP-21065L. Like the ADSP-2116x family, the ADSP-2136x family is based on the original ADSP-2106x SHARC family. The ADSP-2136x preserves much of the ADSP-2106x architecture and is code compatible to the ADSP-21160, while extending performance and functionality. For background information on SHARC and the ADSP-2106x Family processors, see the *ADSP-2106x SHARC User's Manual*.

Processor Core Enhancements

Computational bandwidth on the ADSP-2136x processors is significantly greater than that on the ADSP-2106x processors. The increase comes from raising the operational frequency and adding another processing element: ALU, shifter, multiplier, and register file. The new processing element lets the processor process multiple data streams in parallel (SIMD mode). The ADSP-2136x processors operate at up to 400 MHz using a five stage pipeline.

The program sequencer has several enhancements: new interrupt vector table definitions, SIMD mode stack and conditional execution model, and instruction decodes associated with new instructions. Interrupt vectors have been added that detect illegal memory accesses. Also, mode stack and mode mask support have been added to improve context switch time.

The data address generators are improved from previous architectures in that DAG2 (for the PM bus) has the same addressing capability as DAG1 (for the DM bus). The DAG registers move 64 bits per cycle. Additionally, the DAGs support the new memory map and long word transfer capability. Circular buffering on the ADSP-2136x processors can be quickly disabled on interrupts and restored on the return. Data "broadcast", from one memory location to both data register files, is determined by appropriate index register usage.

Processor Internal Bus Enhancements

The PM, DM, and I/O data buses have increased from 32 bits on the ADSP-2106x processors to 64 bits. Additional multiplexing and control logic enable 16-, 32-, or 64-bit wide moves between both register files and memory. The ADSP-2136x processors are capable of broadcasting a single memory location to each of the register files in parallel. Also, the ADSP-2136x processors permit register contents to be exchanged between the two processing elements' register files in a single cycle.

Memory Organization Enhancements

The ADSP-2136x processors memory maps differ from the memory map of the ADSP-2106x processor. The system memory map on each processor group supports double-word transfers each cycle, reflects extended internal memory capacity for derivative designs, and works with an updated control register for SIMD support. The ADSP-2136x processor family provides enough on-chip memory for several audio decoders.

JTAG Port Enhancements

The JTAG port differs from the JTAG port of the ADSP-2106x processors. The ADSP-2136x processors offer ROM-based security. These security features prevent piracy of codes and algorithms and prohibit inspection of on-chip memory via the emulator or buses. The JTAG port uses program controls to limit access to sensitive code in memory. An assigned 64-bit key must be used to access protected memory regions.

The background telemetry channel (BTC) allows the emulator to feed new data to the processor. It also gets updates from the processor in real time. By using this function (that operates in the background), programmers can read and write data to a set of memory-mapped buffers that are accessible by the emulator while the core is running.

Instruction Set Enhancements

The ADSP-2136x processors provide source code compatibility with the previous SHARC processor family members, to the application assembly source code level. All instructions, control registers, and system resources available in the ADSP-2106x core programming model are also available in the ADSP-2136x processors. Instructions, control registers, or other facilities, required to support the new feature set of the ADSP-2136x core include:

- Code compatibility with the ADSP-21160 SIMD core
- Supersets of the ADSP-2106x programming model
- Reserved facilities in the ADSP-2106x programming model
- Symbol name changes from the ADSP-2106x and ADSP-2136x processor programming models

These name changes can be managed through reassembly by using the ADSP-2136x development tools to apply the ADSP-2136x symbol definitions header file and linker description file. While these changes have no direct impact on existing core applications, system and I/O processor initialization code and control code do require modifications.

Although the porting of source code written for the ADSP-2106x family to the ADSP-2136x has been simplified, code changes are required to take full advantage of the new ADSP-2136x processor features. For more information, see "Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference.

2 PROCESSING ELEMENTS

The processor's processing elements (PEx and PEy) perform numeric processing for processor algorithms. Each processing element contains a data register file and three computation units—an arithmetic/logic unit (ALU), a multiplier, and a shifter. Computational instructions for these elements include both fixed-point and floating-point operations, and each computational instruction executes in a single cycle.

The computational units in a processing element handle different types of operations. The ALU performs arithmetic and logic operations on fixed-point and floating-point data. The multiplier performs floating-point and fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations. The shifter computes logical shifts, arithmetic shifts, bit manipulation, field deposit, and field extraction operations on 32-bit operands. The shifter can also derive exponents.

Data flow paths through the computational units are arranged in parallel, as shown in Figure 2-1. The output of any computational unit may serve as the input of any computational unit on the next instruction cycle. Data moving in and out of the computational units goes through a 10-port register file, consisting of 16 primary registers and 16 alternate registers. Two ports on the register file connect to the PM and DM data buses, allowing data transfer between the computational units and memory (and anything else) connected to these buses.

The processor's assembly language provides access to the data register files in both processing elements. The syntax allows programs to move data to and from these registers, specify a computation's data format and provide naming conventions for the registers, all at the same time. For information on the data register names, see "Data Register File" on page 2-37.

Figure 2-1 provides a graphical guide to the other topics in this chapter. First, a description of the `MODE1` register shows how to set rounding, data format, and other modes for the processing elements. The dashed box indicates which components can be controlled by the `MODE1` register. Next, an examination of each computational unit provides details on operation and a summary of computational instructions. Outside the computational units, details on register files and data buses identify how to flow data for computations. Finally, details on the processor's advanced parallelism reveal how to take advantage of multifunction instructions and single-instruction, multiple-data (SIMD) mode.

Numeric Formats

The processor supports the 32-bit single-precision floating-point data format defined in the IEEE Standard 754/854. In addition, the processor supports an extended-precision version of the same format with eight additional bits in the mantissa (40 bits total). The processor also supports 32-bit fixed-point formats—fractional and integer—which can be signed (two's-complement) or unsigned.

IEEE Single-Precision Floating-Point Data Format

The IEEE Standard 754/854 specifies a 32-bit single-precision floating-point format, shown in Figure 2-2. A number in this format consists of a sign bit(s), a 24-bit significand, and an 8-bit unsigned-magnitude exponent (e).

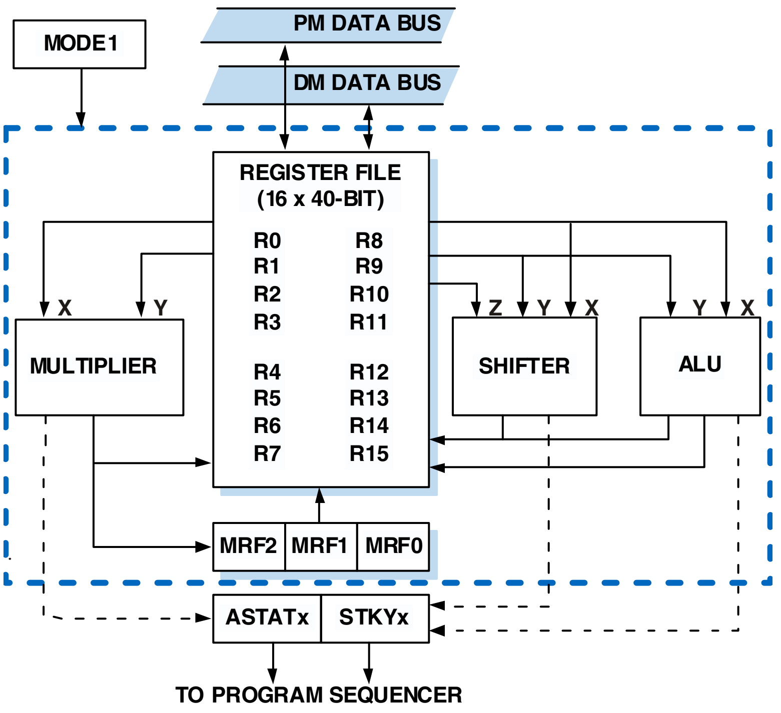


Figure 2-1. Computational Block

For normalized numbers, the significand consists of a 23-bit fraction, *f* and a "hidden" bit of 1 that is implicitly presumed to precede f_{22} in the significand. The binary point is presumed to lie between this hidden bit and f_{22}. The least significant bit (LSB) of the fraction is f_0; the LSB of the exponent is e_0.

The hidden bit effectively increases the precision of the floating-point significand to 24 bits from the 23 bits actually stored in the data format. It also ensures that the significand of any number in the IEEE normalized number format is always greater than or equal to one and less than two.

The unsigned exponent, e, can range between $1 \le e \le 254$ for normal numbers in single-precision format. This exponent is biased by +127 (254, 2). To calculate the true unbiased exponent, subtract 127 from e.

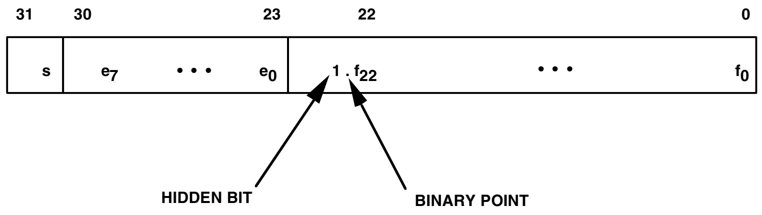


Figure 2-2. IEEE 32-Bit Single-Precision Floating-Point Format

The IEEE Standard also provides several special data types in the single-precision floating-point format:

- An exponent value of 255 (all ones) with a non-zero fraction is a not-a-number (NAN). NANs are usually used as flags for data flow control, for the values of uninitialized variables, and for the results of invalid operations such as $0 * \infty$.
- Infinity is represented as an exponent of 255 and a zero fraction. Note that because the fraction is signed, both positive and negative infinity can be represented.
- Zero is represented by a zero exponent and a zero fraction. As with infinity, both positive zero and negative zero can be represented.

The IEEE single-precision floating-point data types supported by the processor and their interpretations are summarized in Table 2-1.

Table 2-1. IEEE Single-Precision Floating-Point Data Types

Type	Exponent	Fraction	Value
NAN	255	Non-zero	Undefined
Infinity	255	0	$(-1)^s$ Infinity
Normal	$1 \leq e \leq 254$	Any	$(-1)^s (1.f_{22\text{-}0})\ 2^{e-127}$
Zero	0	0 $(-1)^s$ Zero	

Extended-Precision Floating-Point Format

The extended-precision floating-point format is 40 bits wide, with the same 8-bit exponent as in the IEEE standard format but with a 32-bit significand. This format is shown in Figure 2-3. In all other respects, the extended-precision floating-point format is the same as the IEEE standard format.

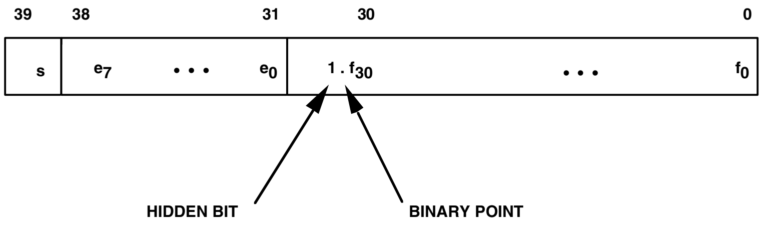


Figure 2-3. 40-Bit Extended-Precision Floating-Point Format

Short Word Floating-Point Format

The processor supports a 16-bit floating-point data type and provides conversion instructions for it. The short float data format has an 11-bit mantissa with a 4-bit exponent plus sign bit, as shown in Figure 2-4. The 16-bit floating-point numbers reside in the lower 16 bits of the 32-bit floating-point field.

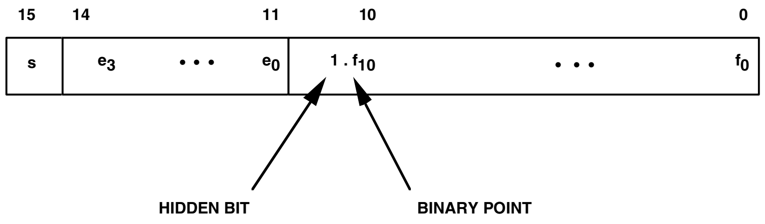


Figure 2-4. 16-Bit Floating-Point Format

Packing for Floating-Point Data

Two shifter instructions, `FPACK` and `FUNPACK`, perform the packing and unpacking conversions between 32-bit floating-point words and 16-bit floating-point words. The `FPACK` instruction converts a 32-bit IEEE floating-point number to a 16-bit floating-point number. The `FUNPACK` instruction converts 16-bit floating-point numbers back to 32-bit IEEE floating-point. Each instruction executes in a single cycle. The results of the `FPACK` and `FUNPACK` operations appear in Table 2-2 and Table 2-3.

Table 2-2. FPACK Operations

Condition	Result
135 < exp	Largest magnitude representation.
120 < exp ≤ 135	Exponent is most significant bit (MSB) of source exponent concatenated with the three least significant bits (LSBs) of source exponent. The packed fraction is the rounded upper 11 bits of the source fraction.
109 < exp ≤ 120	Exponent = 0. Packed fraction is the upper bits (source exponent – 110) of the source fraction prefixed by zeros and the "hidden" one. The packed fraction is rounded.
exp < 110	Packed word is all zeros.
exp = source exponent **sign bit remains the same in all cases**	

Table 2-3. FUNPACK Operations

Condition	Result
0 < exp ≤ 15	Exponent is the 3 LSBs of the source exponent prefixed by the MSB of the source exponent and four copies of the complement of the MSB. The unpacked fraction is the source fraction with 12 zeros appended.
exp = 0	Exponent is (120 – N) where N is the number of leading zeros in the source fraction. The unpacked fraction is the remainder of the source fraction with zeros appended to pad it and the "hidden" one stripped away.
exp = source exponent **sign bit remains the same in all cases**	

The short float type supports gradual underflow. This method sacrifices precision for dynamic range. When packing a number which would have underflowed, the exponent is set to zero and the mantissa (including *hidden* 1) is right-shifted the appropriate amount. The packed result is a denormal, which can be unpacked into a normal IEEE floating-point number.

During the FPACK operation, an overflow sets the SV condition and non-overflow clears it. During the FUNPACK operation, the SV condition is cleared. The SZ and SS conditions are cleared by both instructions.

Fixed-Point Formats

The processor supports two 32-bit fixed-point formats—fractional and integer. In both formats, numbers can be signed (two's-complement) or unsigned. The four possible combinations are shown in Figure 2-5. In the fractional format, there is an implied binary point to the left of the most significant magnitude bit. In integer format, the binary point is understood to be to the right of the LSB. Note that the sign bit is negatively weighted in a two's-complement format.

If one operand is signed and the other unsigned, the result is signed. If both inputs are signed, the result is signed and automatically shifted left one bit. The LSB becomes zero and bit 62 moves into the sign bit position. Normally bit 63 and bit 62 are identical when both operands are signed. (The only exception is full-scale negative multiplied by itself.) Thus, the left-shift normally removes a redundant sign bit, increasing the precision of the most significant product. Also, if the data format is fractional, a single bit left-shift renormalizes the MSP to a fractional format. The signed formats with and without left-shifting are shown in Figure 2-7.

ALU outputs have the same width and data format as the inputs. The multiplier, however, produces a 64-bit product from two 32-bit inputs. If both operands are unsigned integers, the result is a 64-bit unsigned integer. If both operands are unsigned fractions, the result is a 64-bit unsigned fraction. These formats are shown in Figure 2-6.

The multiplier has an 80-bit accumulator to allow the accumulation of 64-bit products. For more information on the multiplier and accumulator, see "Multiply Accumulator (Multiplier)" on page 2-22.

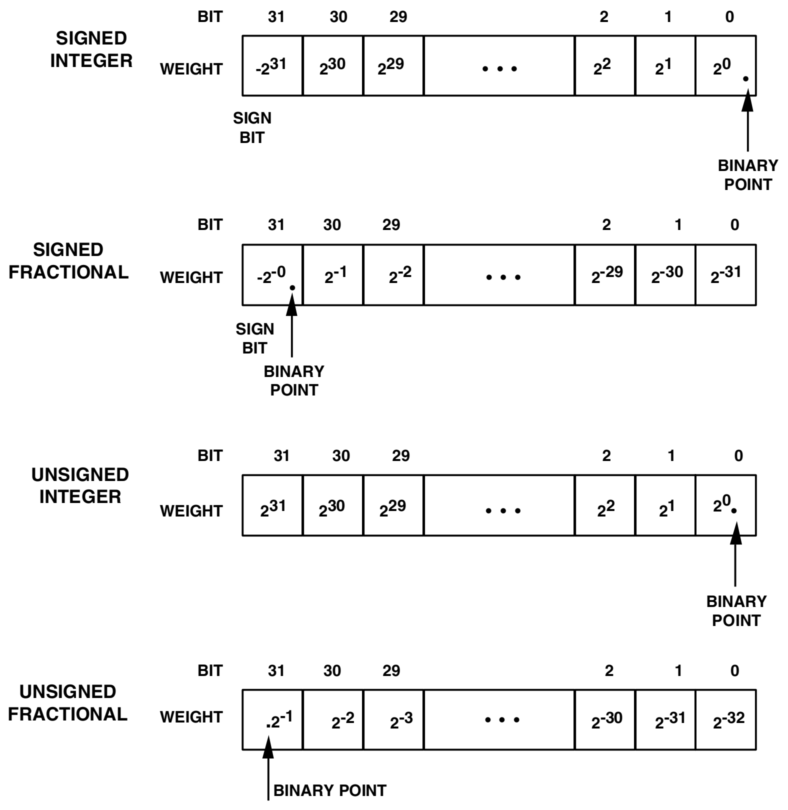

SIGNED INTEGER
BIT 31 30 29 2 1 0
WEIGHT -2^{31} 2^{30} 2^{29} • • • 2^{2} 2^{1} 2^{0}
SIGN BIT
BINARY POINT
SIGNED FRACTIONAL
BIT 31 30 29 2 1 0
WEIGHT -2^{-0} 2^{-1} 2^{-2} • • • 2^{-29} 2^{-30} 2^{-31}
SIGN BIT
BINARY POINT
UNSIGNED INTEGER
BIT 31 30 29 2 1 0
WEIGHT 2^{31} 2^{30} 2^{29} • • • 2^{2} 2^{1} 2^{0}
BINARY POINT
UNSIGNED FRACTIONAL
BIT 31 30 29 2 1 0
WEIGHT $.2^{-1}$ 2^{-2} 2^{-3} • • • 2^{-30} 2^{-31} 2^{-32}
BINARY POINT

Figure 2-5. 32-Bit Fixed-Point Formats

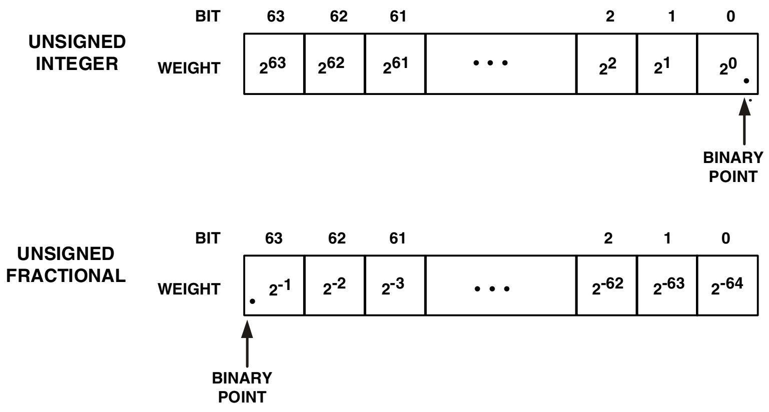


Figure 2-6. 64-Bit Unsigned Fixed-Point Product

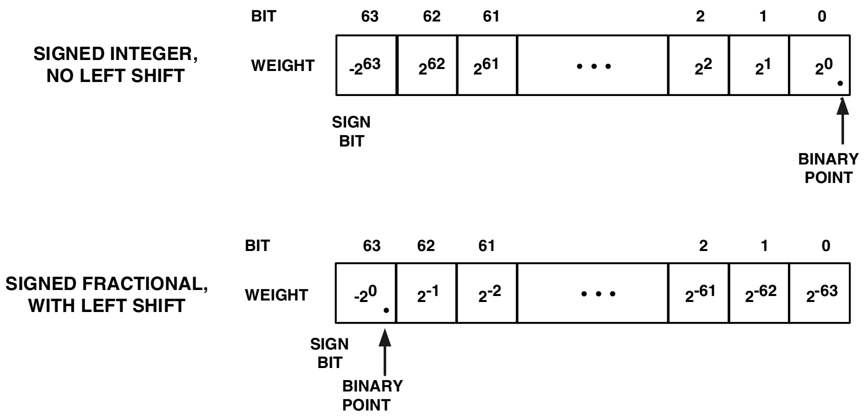


Figure 2-7. 64-Bit Signed Fixed-Point Product

Setting Computational Modes

The MODE1 register controls the operating mode of the processing elements. Table B-2 on page B-5 lists the bits in the MODE1 register. The following MODE1 bits control computational modes:

- **Floating-point data format.** Bit 16 (RND32) rounds floating-point data to 32 bits (if 1) or rounds to 40 bits (if 0).
- **Rounding mode.** Bit 15 (TRUNC) rounds results with round-to-zero (if 1) or round-to-nearest (if 0).
- **ALU saturation.** Bit 13 (ALUSAT) saturates results on positive or negative fixed-point overflows (if 1) or returns unsaturated results (if 0).
- **Short word sign extension.** Bit 14 (SSE) sign extends short word 16-bit data (if 1) or zero-fill the upper 16 bits (if 0).
- **Secondary processor element (PEy).** Bit 21 (PEYEN) enables computations in PEy (SIMD mode) (if 1) or disables PEy (SISD mode) (if 0).

32-Bit Floating-Point Format (Normal Word)

In the default mode, (RND32 bit=1), the multiplier and ALU support a single-precision floating-point format, which is specified in the IEEE 754/854 standard. For more information on this standard, see "Numeric Formats" on page 2-2. This format is IEEE 754/854 compatible for single-precision floating-point operations in all respects except:

- The processor does not provide inexact flags. An inexact flag is an exception flag whose bit position is inexact. The inexact exception occurs if the rounded result of an operation is not identical to the exact (infinitely precise) result. Thus, an inexact exception always occurs when an overflow or an underflow occurs.
- NAN (Not-A-Number) inputs generate an invalid exception and return a quiet NAN (all 1s).
- Denormal operands, using denormalized (or tiny) numbers, flush to zero when input to a computational unit and do not generate an underflow exception. A denormal operand is one of the floating-point operands with an absolute value too small to represent with full precision in the significant. The denormal exception occurs if one or more of the operands is a denormal number. This exception is never regarded as an error.
- The processor supports round-to-nearest and round-toward-zero modes, but does not support round to +infinity and round to –infinity.

IEEE single-precision floating-point data uses a 23-bit mantissa with an 8-bit exponent plus sign bit. In this case, the computation unit sets the eight LSBs of floating-point inputs to zeros before performing the operation. The mantissa of a result rounds to 23 bits (not including the hidden bit), and the 8 LSBs of the 40-bit result clear to zeros to form a 32-bit number, which is equivalent to the IEEE standard result.

In fixed-point to floating-point conversion, the rounding boundary is always 40 bits, even if the `RND32` bit is set.

40-Bit Floating-Point Format

In extended-precision mode (`RND32` bit=0), the processor supports a 40-bit extended-precision floating-point mode, which has eight additional LSBs of the mantissa and is compliant with the 754/854 standards. However, results in this format are more precise than the IEEE single-precision standard specifies. Extended-precision floating-point data uses a 31-bit mantissa with a 8-bit exponent plus sign a bit.

16-Bit Floating-Point Format (Short Word)

The processor supports a 16-bit floating-point storage format and provides instructions that convert the data for 40-bit computations. The 16-bit floating-point format uses an 11-bit mantissa with a 4-bit exponent plus sign bit. The 16-bit data goes into bits 23 through 8 of a data register. Two shifter instructions, `FPACK` and `FUNPACK`, perform the packing and unpacking conversions between 32-bit floating-point words and 16-bit floating-point words. The `FPACK` instruction converts a 32-bit IEEE floating-point number in a data register into a 16-bit floating-point number. `FUNPACK` converts a 16-bit floating-point number in a data register to a 32-bit IEEE floating-point number. Each instruction executes in a single cycle.

When 16-bit data is written to bits 23 through 8 of a data register, the processor automatically extends the data into a 32-bit integer (bits 39 through 8). If the `SSE` bit in `MODE1` is set (1), the processor sign-extends the upper 16 bits. If the `SSE` bit is cleared (0), the processor zeros the upper 16 bits.

The 16-bit floating-point format supports gradual underflow. This method sacrifices precision for dynamic range. When packing a number that would have underflowed, the exponent clears to zero and the mantissa (including a “hidden” 1) right-shifts the appropriate amount. The packed result is a denormal, which can be unpacked into a normal IEEE floating-point number.

32-Bit Fixed-Point Format

The processor represents fixed-point numbers in 32 bits, occupying the 32 MSBs in 40-bit data registers. Fixed-point data may be fractional or integer numbers and unsigned or two’s-complement. Each computational unit has limitations on how these formats may be mixed for a given operation. All computational units read the upper 32 bits of data (inputs, operands) from the 40-bit registers (ignoring the eight LSBs) and write results to the upper 32 bits (zeroing the eight LSBs).

Rounding Mode

The `TRUNC` bit in the `MODE1` register determines the rounding mode for all ALU operations, all floating-point multiplies, and fixed-point multiplies of fractional data. The processor supports two rounding modes—round-toward-zero and round-toward-nearest. The rounding modes comply with the IEEE 754 standard and have the following definitions:

- Round-toward-zero (`TRUNC` bit=1). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to zero. This is equivalent to truncation.

- Round-toward-nearest (`TRUNC` bit=0). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to the result before rounding. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is the number that has an LSB equal to zero.

Statistically, rounding up occurs as often as rounding down, so there is no large sample bias. Because the maximum floating-point value is one LSB less than the value that represents infinity, a result that is halfway between the maximum floating-point value and infinity rounds to infinity in this mode.

Though these rounding modes comply with standards set for floating-point data, they also apply for fixed-point multiplier operations on fractional data. The same two rounding modes are supported, but only the round-to-nearest operation is actually performed by the multiplier. Using its local result register for fixed-point operations, the multiplier rounds-to-zero by reading only the upper bits of the result and discarding the lower bits.

Using Computational Status

The multiplier and ALU each provide exception information when executing floating-point operations. Each unit updates overflow, underflow, and invalid operation flags in the processing element's arithmetic status (ASTATx and ASTATy) registers and sticky status (STKYx and STKYy) registers. An underflow, overflow, or invalid operation from any unit also generates a maskable interrupt. There are three ways to use floating-point exceptions from computations in program sequencing:

- Enable interrupts and use an interrupt service routine (ISR) to handle the exception condition immediately. This method is appropriate if it is important to correct all exceptions as they occur.
- Use conditional instructions to test the exception flags in the ASTATx or ASTATy registers after the instruction executes. This method permits monitoring each instruction's outcome.

- Use the bit test (BTST) instruction to examine exception flags in the STKY register after a series of operations. If any flags are set, some of the results are incorrect. Use this method when exception handling is not critical.

More information on ASTAT and STKY status appears in the sections that describe the computational units. For summaries relating instructions and status bits, see Table 2-4, Table 2-5, Table 2-7, Table 2-9, and Table 2-10.

Arithmetic Logic Unit (ALU)

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

- Floating-point addition, subtraction, add/subtract, average
- Fixed-point addition, subtraction, add/subtract, average
- Floating-point manipulation: binary log, scale, mantissa
- Fixed-point add with carry, subtract with borrow, increment, decrement
- Logical And, Or, Xor, Not
- Functions: ABS, PASS, MIN, MAX, CLIP, COMPARE
- Format conversion
- Reciprocal and reciprocal square root primitives

ALU Operation

ALU instructions take one or two inputs: X input and Y input. These inputs (known as operands) can be any data registers in the register file. Most ALU operations return one result; in add/subtract operations, the ALU operation returns two results; in compare operations, the ALU operation returns no result (only flags are updated). ALU results can be returned to any location in the register file.

Because of the 5-stage pipeline in the ADSP-2136x processor core, the operands are fetched before the results are written back. Therefore, the ALU can read and write the same register file location in a single cycle. If the ALU operation is fixed-point, the inputs are treated as 32-bit fixed-point operands. The ALU transfers the upper 32 bits from the source location in the register file. For fixed-point operations, the result(s) are 32-bit fixed-point values. Some floating-point operations (`LOGB`, `MANT` and `FIX`) can also yield fixed-point results.

The processor transfers fixed-point results to the upper 32 bits of the data register and clears the lower eight bits of the register. The format of fixed-point operands and results depends on the operation. In most arithmetic operations, there is no need to distinguish between integer and fractional formats. Fixed-point inputs to operations such as scaling a floating-point value are treated as integers. For purposes of determining status such as overflow, fixed-point arithmetic operands and results are treated as two's-complement numbers.

ALU Saturation

When the `ALUSAT` bit is set (=1) in the `MODE1` register, the ALU is in saturation mode. In this mode, positive fixed-point overflows return the maximum positive fixed-point number (0x7FFF FFFF), and negative overflows return the maximum negative number (0x8000 0000).

When the ALUSAT bit is cleared (=0) in the MODE1 register, fixed-point results that overflow are not saturated; the upper 32 bits of the result are returned unaltered.

ALU Status Flags

ALU operations update seven status flags in the processing element's arithmetic status (ASTATx and ASTATy) registers. Table B-4 on page B-14 lists the bits in these registers. The following bits in ASTATx or ASTATy registers flag the ALU status (a 1 indicates the condition) of the most recent ALU operation:

- ALU result zero or floating-point underflow, bit 0 (AZ)
- ALU overflow, bit 1 (AV)
- ALU result negative, bit 2 (AN)
- ALU fixed-point carry, bit 3 (AC)
- ALU X input sign for ABS, MANT operations, bit 4 (AS)
- ALU floating-point invalid operation, bit 5 (AI)
- Last ALU operation was a floating-point operation, bit 10 (AF)
- Compare accumulation register results of last eight compare operations, bits 31-24 (CACC)

ALU operations also update four sticky status flags in the processing element's sticky status (STKYx and STKYy) registers. Table B-5 on page B-20 lists the bits in these registers. The following bits in STKYx or STKYy flag the ALU status (a 1 indicates the condition). Once set, a sticky flag remains high until explicitly cleared:

- ALU floating-point underflow, bit 0 (AUS)
- ALU floating-point overflow, bit 1 (AVS)

- ALU fixed-point overflow, bit 2 (AOS)
- ALU floating-point invalid operation, bit 5 (AIS)

Flag updates occur at the end of the cycle in which the status is generated and is available on the next cycle. If a program writes the arithmetic status register or sticky status register explicitly in the same cycle that the ALU is performing an operation, the explicit write to the status register supersedes any flag update from the ALU operation.

ALU Instruction Summary

Table 2-4 and Table 2-5 list the ALU instructions and show how they relate to ASTATx,y and STKYx,y flags. For more information on assembly language syntax, see "Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference. In these tables, note the meaning of these symbols:

- Rn, Rx, Ry indicate any register file location; treated as fixed-point
- Fn, Fx, Fy indicate any register file location; treated as floating-point
- * indicates that the flag may be set or cleared, depending on the results of instruction
- ** indicates that the flag may be set (but not cleared), depending on the results of the instruction
- – indicates no effect

Table 2-4. Fixed-Point ALU Instruction Summary

Instruction	**ASTATx,y Status Flags**							**STKYx,y Status Flags**				
Fixed-Point:	**AZ**	**AV**	**AN**	**AC**	**AS**	**AI**	**AF**	**CACC**	**AUS**	**AVS**	**AOS**	**AIS**
Rn = Rx + Ry	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx – Ry	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx + Ry + CI	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx – Ry + CI – 1	*	*	*	*	0	0	0	–	–	–	**	–
Rn = (Rx + Ry)/2	*	0	*	*	0	0	0	–	–	–	–	–
COMP(Rx, Ry)	*	0	*	0	0	0	0	*	–	–	–	–
COMPU(Rx,Ry)	*	0	*	0	0	0	0	*	--	--	--	--
Rn = Rx + CI	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx + CI – 1	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx + 1	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx – 1	*	*	*	*	0	0	0	–	–	–	**	–
Rn = –Rx	*	*	*	*	0	0	0	–	–	–	**	–
Rn = ABS Rx	*	*	0	0	*	0	0	–	–	–	**	–
Rn = PASS Rx	*	0	*	0	0	0	0	–	–	–	–	–
Rn = Rx AND Ry	*	0	*	0	0	0	0	–	–	–	–	–
Rn = Rx OR Ry	*	0	*	0	0	0	0	–	–	–	–	–
Rn = Rx XOR Ry	*	0	*	0	0	0	0	–	–	–	–	–
Rn = NOT Rx	*	0	*	0	0	0	0	–	–	–	–	–
Rn = MIN(Rx, Ry)	*	0	*	0	0	0	0	–	–	–	–	–
Rn = MAX(Rx, Ry)	*	0	*	0	0	0	0	–	–	–	–	–
Rn = CLIP Rx BY Ry	*	0	*	0	0	0	0	–	–	–	–	–

Table 2-5. Floating-Point ALU Instruction Summary

Instruction	ASTATx,y Status Flags							STKYx,y Status Flags				
Floating-Point:	AZ	AV	AN	AC	AS	AI	AF	CACC	AUS	AVS	AOS	AIS
Fn = Fx + Fy	*	*	*	0	0	*	1	–	**	**	–	**
Fn = Fx – Fy	*	*	*	0	0	*	1	–	**	**	–	**
Fn = ABS (Fx + Fy)	*	*	0	0	0	*	1	–	**	**	–	**
Fn = ABS (Fx – Fy)	*	*	0	0	0	*	1	–	**	**	–	**
Fn = (Fx + Fy)/2	*	0	*	0	0	*	1	–	**	–	–	**
COMP(Fx, Fy)	*	0	*	0	0	*	1	*	–	–	–	**
Fn = –Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn = ABS Fx	*	*	0	0	*	*	1	–	–	**	–	**
Fn = PASS Fx	*	0	*	0	0	*	1	–	–	–	–	**
Fn = RND Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn = SCALB Fx BY Ry	*	*	*	0	0	*	1	–	**	**	–	**
Rn = MANT Fx	*	*	0	0	*	*	1	–	–	**	–	**
Rn = LOGB Fx	*	*	*	0	0	*	1	–	–	**	–	**
Rn = FIX Fx BY Ry	*	*	*	0	0	*	1	–	**	**	–	**
Rn = FIX Fx	*	*	*	0	0	*	1	–	**	**	–	**
Fn = FLOAT Rx BY Ry	*	*	*	0	0	0	1	–	**	**	–	–
Fn = FLOAT Rx	*	0	*	0	0	0	1	–	–	–	–	–
Fn = RECIPS Fx	*	*	*	0	0	*	1	–	**	**	–	**
Fn = RSQRTS Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn = Fx COPYSIGN Fy	*	0	*	0	0	*	1	–	–	–	–	**
Fn = MIN(Fx, Fy)	*	0	*	0	0	*	1	–	–	–	–	**
Fn = MAX(Fx, Fy)	*	0	*	0	0	*	1	–	–	–	–	**
Fn = CLIP Fx BY Fy	*	0	*	0	0	*	1	–	–	–	–	**

Multiply Accumulator (Multiplier)

The multiplier performs fixed-point or floating-point multiplication and fixed-point multiply/accumulate operations. Fixed-point multiply/accumulates are available with cumulative addition or cumulative subtraction. Multiplier floating-point instructions operate on 32-bit or 40-bit floating-point operands and output 32-bit or 40-bit floating-point results. Multiplier fixed-point instructions operate on 32-bit fixed-point data and produce 80-bit results. Inputs are treated as fractional or integer, unsigned or two's-complement. Multiplier instructions include:

- Floating-point multiplication
- Fixed-point multiplication
- Fixed-point multiply/accumulate with addition, rounding optional
- Fixed-point multiply/accumulate with subtraction, rounding optional
- Rounding multiplier result register
- Saturating multiplier result register
- Clearing multiplier result register

Multiplier Operation

The multiplier takes two inputs: X and Y. These inputs (also known as operands) can be any data registers in the register file. The multiplier can accumulate fixed-point results in the local multiplier result (`MRF`) registers or write results back to the register file. The results in `MRF` can also be rounded or saturated in separate operations. Floating-point multiplies yield floating-point results, which the multiplier writes directly to the register file.

Because of the 5-stage pipeline in the ADSP-2136x processor core, the operands are fetched before the results are written back. Therefore, the multiplier can read and write the same register file location in a single cycle.

For fixed-point multiplies, the multiplier reads the inputs from the upper 32 bits of the data registers. Fixed-point operands may be integer, fractional or both formats. The format of the result matches the format of the inputs. Each fixed-point operand may be either an unsigned number or a two's-complement number. If both inputs are fractional and signed, the multiplier automatically shifts the result left one bit to remove the redundant sign bit. The register name(s) within the multiplier instruction specify input data type(s)—Fx for floating-point and Rx for fixed-point.

Multiplier Result Register (Fixed-Point)

Fixed-point operations place 80-bit results in the multiplier's foreground MRF register or background MRB register, depending on which is active. For more information on selecting the result register, see "Alternate (Secondary) Data Registers" on page 2-39.

The location of a result in the MRF register's 80-bit field depends on whether the result is in fractional or integer format, as shown in Figure 2-8. If the result is sent directly to a data register, the 32-bit result with the same format as the input data is transferred, using bits 63-32 for a fractional result or bits 31-0 for an integer result. The eight LSBs of the 40-bit register file location are zero-filled.

Fractional results can be rounded-to-nearest before being sent to the register file. If rounding is not specified, discarding bits 31-0 effectively truncates a fractional result (rounds to zero). For more information on rounding, see "Rounding Mode" on page 2-14.

The MRF register is comprised of the MRF2, MRF1, and MRF0 registers, which individually can be read from or written to the register file. Each of these registers has the same format. When data is read from MRF2, it is

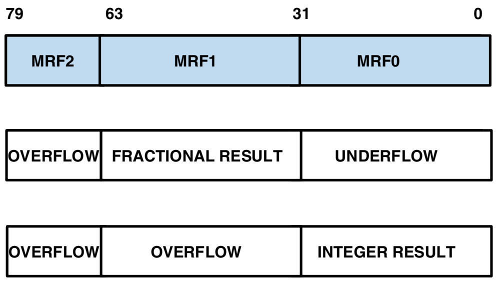


Figure 2-8. Multiplier Fixed-Point Result Placement

sign-extended to 32 bits as shown in Figure 2-9. The processor zero-fills the eight LSBs of the 40-bit register file location when data is read from `MRF2`, `MRF1`, or `MRF0` written to the register file. When the processor writes data into `MRF2`, `MRF1`, or `MRF0` from the 32 MSBs of a register file location, the eight LSBs are ignored. Data written to `MRF1` is sign-extended to `MRF2`, repeating the MSB of `MRF1` in the 16 bits of `MRF2`. Data written to `MRF0` is not sign-extended.

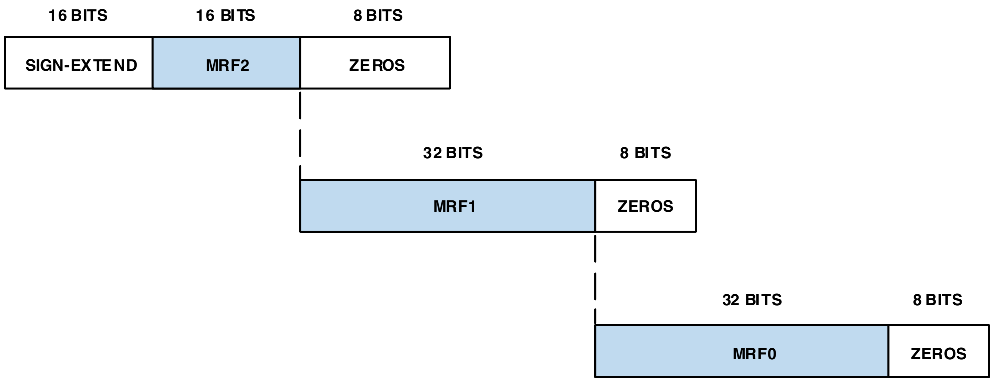


Figure 2-9. MR Transfer Formats

In addition to multiply, fixed-point operations include accumulate, round, and saturate fixed-point data. There are three `MRF` register operations: clear (`CLR`), round (`RND`), and saturate (`SAT`).

The `CLR` operation (`MRF=0`) resets the specified `MRF` register to zero. Often, it is best to perform this operation at the start of a multiply/accumulate operation to remove the results of the previous operation.

The `RND` operation (`MRF=RND MRF`) applies only to fractional results and integer results are not effected. This operation rounds the 80-bit `MRF` value to nearest at bit 32, for example, the `MRF1-MRF0` boundary. Rounding a fixed-point result occurs as part of a multiply or multiply/accumulate operation or as an explicit operation on the `MRF` register. The rounded result in `MRF1` can be sent to the register file or back to the same `MRF` register. To round a fractional result to zero (truncation) instead of to nearest, a program transfers the unrounded result from `MRF1`, discarding the lower 32 bits in `MRF0`.

The `SAT` operation (`MRF=SAT MRF`) sets `MRF` to a maximum value if the `MRF` value has overflowed. Overflow occurs when the `MRF` value is greater than the maximum value for the data format—unsigned or two's-complement and integer or fractional—as specified in the saturate instruction. The six possible maximum values appear in Table 2-6. The result from `MRF` saturation can be sent to the register file or back to the same `MRF` register.

Table 2-6. Fixed-Point Format Maximum Values (Saturation)

Maximum Number	**(Hexadecimal)**		
	MRF2	**MRF1**	**MRF0**
Two's-complement fractional (positive)	0000	7FFF FFFF	FFFF FFFF
Two's-complement fractional (negative)	FFFF	8000 0000	0000 0000
Two's-complement integer (positive)	0000	0000 0000	7FFF FFFF
Two's-complement integer (negative)	FFFF	FFFF FFFF	8000 0000

Table 2-6. Fixed-Point Format Maximum Values (Saturation) (Cont'd)

Maximum Number	(Hexadecimal)		
	MRF2	MRF1	MRF0
Unsigned fractional number	0000	FFFF FFFF	FFFF FFFF
Unsigned integer number	0000	0000 0000	FFFF FFFF

Multiplier Status Flags

Multiplier operations update four status flags in the processing element's arithmetic status registers (`ASTATx` and `ASTATy`). "Arithmetic Status Registers (ASTATx and ASTATy)" on page B-12 lists the bits in these registers. The bits in the `ASTATx` or `ASTATy` registers that indicate the multiplier status (a 1 indicates the condition) of the most recent multiplier operation are:

- Multiplier result negative, bit 6 (`MN`)
- Multiplier overflow, bit 7 (`MV`)
- Multiplier underflow, bit 8 (`MU`)
- Multiplier floating-point invalid operation, bit 9 (`MI`)

Multiplier operations also update four "sticky" status flags in the processing element's sticky status (`STKYx` and `STKYy`) registers. Table B-5 on page B-20 lists the bits in these registers. Once set, a sticky flag remains high until explicitly cleared. The bits in the `STKYx` or `STKYy` registers that indicate multiplier status (a 1 indicates the condition) are:

- Multiplier fixed-point overflow, bit 6 (`MOS`)
- Multiplier floating-point overflow, bit 7 (`MVS`)
- Multiplier underflow, bit 8 (`MUS`)
- Multiplier floating-point invalid operation, bit 9 (`MIS`)

Flag updates occur at the end of the cycle in which the status is generated and are available on the next cycle. If a program writes the arithmetic status register or sticky register explicitly in the same cycle that the multiplier is performing an operation, the explicit write to ASTAT or STKY supersedes any flag update from the multiplier operation.

Multiplier Instruction Summary

Table 2-7 and Table 2-9 list the multiplier instructions and describe how they relate to ASTATx,y and STKYx,y flags. For more information on assembly language syntax, see "Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference. In these tables, note the meaning of the following symbols:

- Rn, Rx, Ry indicate any register file location; treated as fixed-point
- Fn, Fx, Fy indicate any register file location; treated as floating-point
- * indicates that the flag may be set or cleared, depending on results of instruction
- ** indicates that the flag may be set (but not cleared), depending on results of instruction
- – indicates no effect
- The Input Mods column indicates the types of optional modifiers that can be applied to the instruction inputs. For a list of modifiers, see Table 2-8.

Table 2-7. Fixed-Point Multiplier Instruction Summary

Instruction		ASTATx,y Flags				STKYx,y Flags			
Fixed-Point: For Input Mods, see Table 2-8	Input Mods	MU	MN	MV	MI	MUS	MOS	MVS	MIS
Rn = Rx * Ry	1	*	*	*	0	–	**	–	–
MRF = Rx * Ry	1	*	*	*	0	–	**	–	–
MRB = Rx * Ry	1	*	*	*	0	–	**	–	–
Rn = MRF + Rx * Ry	1	*	*	*	0	–	**	–	–
Rn = MRB + Rx * Ry	1	*	*	*	0	–	**	–	–
MRF = MRF + Rx * Ry	1	*	*	*	0	–	**	–	–
MRB = MRB + Rx * Ry	1	*	*	*	0	–	**	–	–
Rn = MRF – Rx * Ry	1	*	*	*	0	–	**	–	–
Rn = MRB – Rx * Ry	1	*	*	*	0	–	**	–	–
MRF = MRF – Rx * Ry	1	*	*	*	0	–	**	–	–
MRB = MRB – Rx * Ry	1	*	*	*	0	–	**	–	–
Rn = SAT MRF	2	*	*	*	0	–	**	–	–
Rn = SAT MRB	2	*	*	*	0	–	**	–	–
MRF = SAT MRF	2	*	*	*	0	–	**	–	–
MRB = SAT MRB	2	*	*	*	0	–	**	–	–
Rn = RND MRF	3	*	*	*	0	–	**	–	–
Rn = RND MRB	3	*	*	*	0	–	**	–	–
MRF = RND MRF	3	*	*	*	0	–	**	–	–
MRB = RND MRB	3	*	*	*	0	–	**	–	–
MRF = 0	–	0	0	0	0	–	–	–	–
MRB = 0	–	0	0	0	0	–	–	–	–
MRxF = Rn	–	0	0	0	0	–	–	–	–
MRxB = Rn	–	0	0	0	0	–	–	–	–
Rn = MRxF	–	0	0	0	0	–	–	–	–
Rn = MRxB	–	0	0	0	0	–	–	–	–

Table 2-8. Input Modifiers for Fixed-Point Multiplier Instruction

Input Mods from Table 2-7	Input Mods—Options For Fixed-Point Multiplier Instructions
	Note the meaning of the following symbols in this table: Signed input — S Unsigned input — U Integer input — I Fractional input — F Fractional inputs, Rounded output — FR Note that (SF) is the default format for one-input operations, and (SSF) is the default format for two-input operations.
1	(SSF), (SSI), (SSFR), (SUF), (SUI), (SUFR), (USF), (USI), (USFR), (UUF), (UUI), or (UUFR)
2	(SF), (SI), (UF), or (UI)
3	(SF) or (UF)

Table 2-9. Floating-Point Multiplier Instruction Summary

Instruction	ASTATx,y Flags				STKYx,y Flags			
Floating-Point:	MU	MN	MV	MI	MUS	MOS	MVS	MIS
Fn = Fx * Fy	*	*	*	*	**	–	**	**

Barrel Shifter (Shifter)

The shifter performs bit-wise operations on 32-bit fixed-point operands. Shifter operations include:

- Shifts and rotates from off-scale left to off-scale right
- Bit manipulation operations, including bit set, clear, toggle, and test
- Bit field manipulation operations, including extract and deposit
- Fixed-point/floating-point conversion operations, including exponent extract, number of leading 1s or 0s

Shifter Operation

The shifter takes one to three inputs: X, Y, and Z. The inputs (known as operands) can be any register in the register file. Within a shifter instruction, the inputs serve as follows.

- The X input provides data that is operated on.
- The Y input specifies shift magnitudes, bit field lengths, or bit positions.
- The Z input provides data that is operated on and updated.

In the following example, `Rx` is the X input, `Ry` is the Y input, and `Rn` is the Z input. The shifter returns one output (`Rn`) to the register file.

```
Rn = Rn OR LSHIFT Rx BY Ry;
```

As shown in Figure 2-10, the shifter fetches input operands from the upper 32 bits of a register file location (bits 39-8) or from an immediate value in the instruction. Because of the 5-stage pipeline in the

ADSP-2136x processor core, the operands are fetched before the results are written back. Therefore, the shifter can read and write the same register file location in a single cycle.

The X input and Z input are always 32-bit fixed-point values. The Y input is a 32-bit fixed-point value or an 8-bit field (shf8), positioned in the register file. These inputs appear in Figure 2-10.

Some shifter operations produce 8 or 6-bit results. As shown in Figure 2-11, the shifter places these results in the shf8 field or the bit6 field and sign-extends the results to 32 bits. The shifter always returns a 32-bit result.

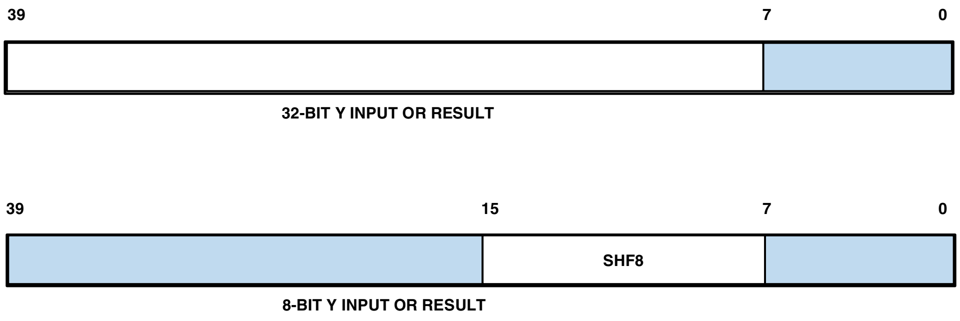


Figure 2-10. Register File Fields for Shifter Instructions

The shifter supports bit field deposit and bit field extract instructions for manipulating groups of bits within an input. The Y input for bit field instructions specifies two 6-bit values, bit6 and len6, which are positioned in the `Ry` register as shown in Figure 2-11. The shifter interprets bit6 and len6 as positive integers. Bit6 is the starting bit position for the deposit or extract, and len6 is the bit field length, which specifies how many bits are deposited or extracted.

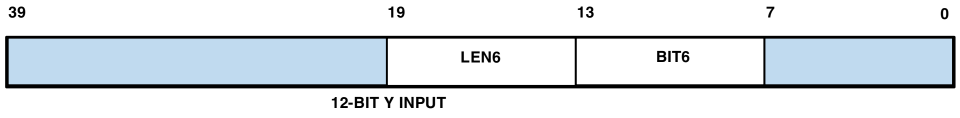


Figure 2-11. Register File Fields for FDEP, FEXT Instructions

Field deposit (`FDEP`) instructions take a group of bits from the input register (starting at the LSB of the 32-bit integer field) and deposit the bits as directed anywhere within the result register. The bit6 value specifies the starting bit position for the deposit. Figure 2-13 shows how the inputs, bit6 and len6, work in a field deposit instruction

```
Rn = FDEP Rx By Ry
```

Figure 2-12 shows bit placement for the following field deposit instruction:

```
R0 = FDEP R1 By R2;
```

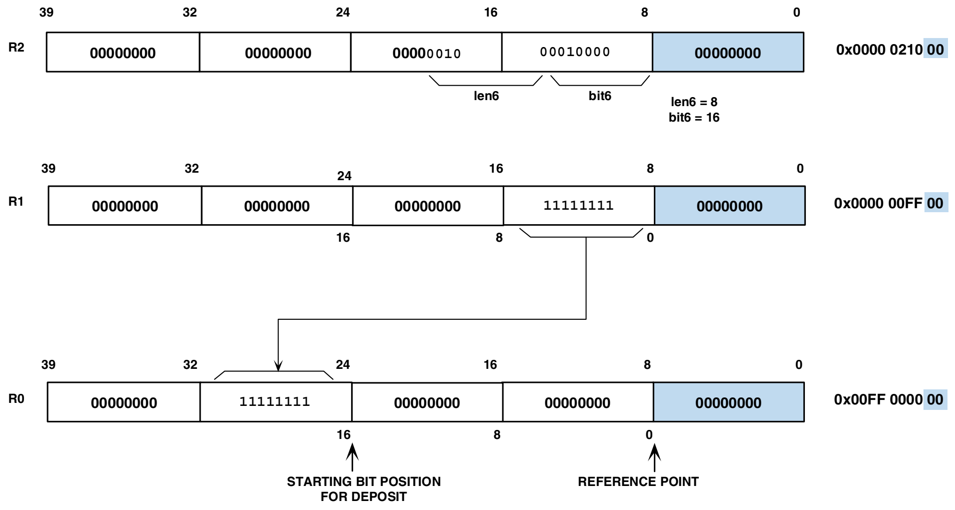


Figure 2-12. Bit Field Deposit Instruction

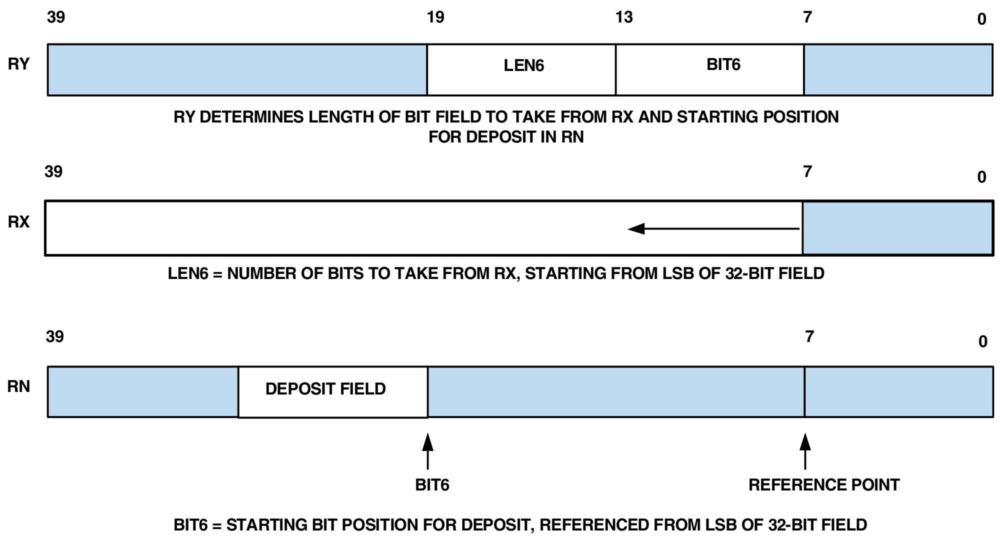

39
19
13
7
0
RY
LEN6
BIT6
RY DETERMINES LENGTH OF BIT FIELD TO TAKE FROM RX AND STARTING POSITION FOR DEPOSIT IN RN
39
7
0
RX
LEN6 = NUMBER OF BITS TO TAKE FROM RX, STARTING FROM LSB OF 32-BIT FIELD
39
7
0
RN
DEPOSIT FIELD
BIT6
REFERENCE POINT
BIT6 = STARTING BIT POSITION FOR DEPOSIT, REFERENCED FROM LSB OF 32-BIT FIELD

Figure 2-13. Bit Field Deposit Instruction

Field extract (`FEXT`) instructions extract a group of bits as directed from anywhere within the input register and place them in the result register, aligned with the LSB of the 32-bit integer field. The bit6 value specifies the starting bit position for the extract.

Figure 2-14 shows bit placement for the following field extract instruction:

```
R3 = FEXT R4 By R5;
```

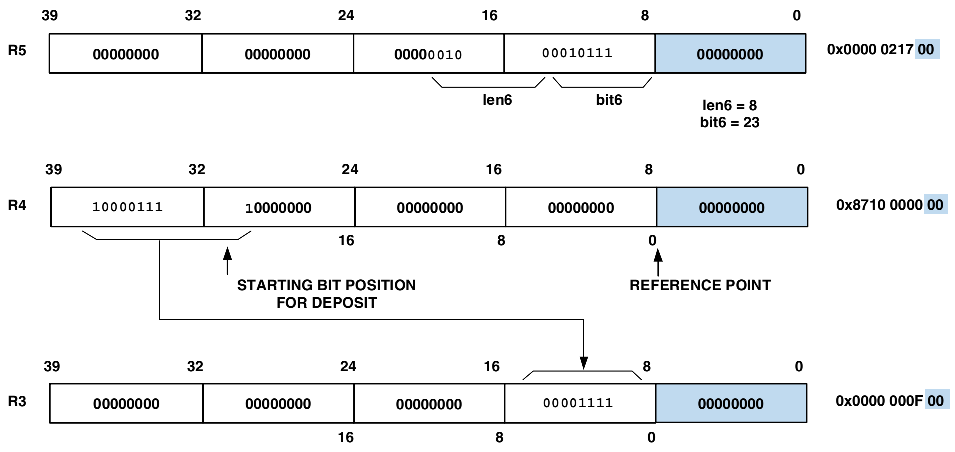


Figure 2-14. Bit Field Extract Instruction

Shifter Status Flags

Shifter operations update three status flags in the processing element's arithmetic status registers (ASTATx and ASTATy). Table B-4 on page B-14 lists the bits in these registers. The following bits in the ASTATx or ASTATy registers indicate shifter status (a 1 indicates the condition) for the most recent ALU operation:

- Shifter overflow of bits to left of MSB, bit 11 (SV)
- Shifter result zero, bit 12 (SZ)
- Shifter input sign for exponent extract only, bit 13 (SS)

A flag update occurs at the end of the cycle in which the status is generated and is available on the next cycle. If a program writes the arithmetic status register explicitly in the same cycle that the shifter is performing an operation, the explicit write to ASTAT supersedes any flag update caused by the shift operation.

Shifter Instruction Summary

Table 2-10 lists the shifter instructions and shows how they relate to `ASTATx,y` flags. For more information on assembly language syntax, see "Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference. In these tables, note the meaning of the following symbols:

- The `Rn`, `Rx`, `Ry` operands indicate any register file location; bit fields used depend on instruction
- The `Fn`, `Fx` operands indicate any register file location; floating-point word
- The `*` symbol indicates that the flag may be set or cleared, depending on data

Table 2-10. Shifter Instruction Summary

Instruction	ASTATx,y Flags		
	SZ	SV	SS
Rn = LSHIFT Rx BY Ry	*	*	0
Rn = LSHIFT Rx BY <data8>	*	*	0
Rn = Rn OR LSHIFT Rx BY Ry	*	*	0
Rn = Rn OR LSHIFT Rx BY <data8>	*	*	0
Rn = ASHIFT Rx BY Ry	*	*	0
Rn = ASHIFT Rx BY<data8>	*	*	0
Rn = Rn OR ASHIFT Rx BY Ry	*	*	0
Rn = Rn OR ASHIFT Rx BY <data8>	*	*	0
Rn = ROT Rx BY Ry	*	0	0
Rn = ROT Rx BY <data8>	*	0	0
Rn = BCLR Rx BY Ry	*	*	0
Rn = BCLR Rx BY <data8>	*	*	0
Rn = BSET Rx BY Ry	*	*	0

Table 2-10. Shifter Instruction Summary (Cont'd)

Instruction	ASTATx,y Flags		
	SZ	SV	SS
Rn = BSET Rx BY <data8>	*	*	0
Rn = BTGL Rx BY Ry	*	*	0
Rn = BTGL Rx BY <data8>	*	*	0
BTST Rx BY Ry	*	*	0
BTST Rx BY <data8>	*	*	0
Rn = FDEP Rx BY Ry	*	*	0
Rn = FDEP Rx BY <bit6>:<len6>	*	*	0
Rn = Rn OR FDEP Rx BY Ry	*	*	0
Rn = Rn OR FDEP Rx BY <bit6>:<len6>	*	*	0
Rn = FDEP Rx BY Ry (SE)	*	*	0
Rn = FDEP Rx BY <bit6>:<len6> (SE)	*	*	0
Rn = Rn OR FDEP Rx BY Ry (SE)	*	*	0
Rn = Rn OR FDEP Rx BY <bit6>:<len6> (SE)	*	*	0
Rn = FEXT Rx BY Ry	*	*	0
Rn = FEXT Rx BY <bit6>:<len6>	*	*	0
Rn = FEXT Rx BY Ry (SE)	*	*	0
Rn = FEXT Rx BY <bit6>:<len6> (SE)	*	*	0
Rn = EXP Rx (EX)	*	0	*
Rn = EXP Rx	*	0	*
Rn = LEFTZ Rx	*	*	0
Rn = LEFTO Rx	*	*	0
Rn = FPACK Fx	0	*	0
Fn = FUNPACK Rx	0	0	0

Data Register File

Each of the processor's processing elements has a data register file, which is a set of data registers that transfers data between the data buses and the computational units. These registers also provide local storage for operands and results.

The two register files consist of 16 primary registers and 16 alternate (secondary) registers. The data registers are 40 bits wide. Within these registers, 32-bit data is left-justified. If an operation specifies a 32-bit data transfer to these 40-bit registers, the eight LSBs are ignored on register reads, and the LSBs are cleared to zeros on writes.

Program memory data accesses and data memory accesses to and from the register file(s) occur on the PM data bus and DM data bus, respectively. One PM data bus access for each processing element and/or one DM data bus access for each processing element can occur in one cycle. Transfers between the register files and the DM or PM data buses can move up to 64 bits of valid data on each bus.

If an operation specifies the same register file location as both an input and output, the 5-stage pipeline fetches the operands before the results are written back. Therefore, the processor uses the old data as the operand, before updating the location with the new result data. If writes to the same location take place in the same cycle, only the write with higher precedence actually occurs. The processor determines precedence for the write operation from the source of the data; from highest to lowest, the precedence is:

1. Data memory or universal register (`Ureg`)
2. Program memory
3. PEx ALU

4. PEy ALU
5. PEx Multiplier
6. PEy Multiplier
7. PEx Shifter
8. PEy Shifter

The data register file in Figure 2-1 on page 2-3 lists register names of R0 through R15 within the PEx's register file. When a program refers to these registers as R0 through R15, the computational units treat the contents of these registers as fixed-point data. To perform floating-point computations, refer to these registers as F0 through F15. For example, the following instructions refer to the same registers, but direct the computational units to perform different operations:

```
F0 = F1 * F2; /* floating-point multiply */

R0 = R1 * R2; /* fixed-point multiply */
```

The *F* and *R* prefixes on register names do not effect the 32-bit or 40-bit data transfer; the naming convention only determines how the ALU, multiplier, and shifter treat the data.

To maintain compatibility with code written for previous SHARC processors, the assembly syntax accommodates references to the PEx and PEy data registers.

Code may refer only to the PEy data registers (S0 through S15) for data move instructions. The rules for using register names are:

- R0 through R15 and F0 through F15 refer to PEx registers for data move and computational instructions, whether the processor is in SISD or SIMD mode.

- R0 through R15 and F0 through F15 refer to both PEx and PEy register for computational instructions in SIMD mode.
- S0 through S15 refer to PEy registers for data move instructions, when the processor is in SISD or SIMD mode.

For more information on SISD and SIMD computational operations, see "Secondary Processing Element (PEy)" on page 2-45. For more information on ADSP-2136x assembly language, see"Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference.

Alternate (Secondary) Data Registers

Each register file has an alternate register set. To facilitate fast context switching, the processor includes alternate register sets for data, results, and data address generator registers. Bits in the MODE1 register control when alternate registers become accessible. While inaccessible, the contents of alternate registers are not affected by processor operations. Note that there is a one cycle latency from the time when writes are made to the MODE1 register until an alternate register set can be accessed. The alternate register sets for data and results are described in this section. For more information on alternate data address generator registers, see "Alternate (Secondary) DAG Registers" on page 4-6.

Bits in the MODE1 register can activate independent alternate data register sets: the lower half (R0-R7 and S0-S7) and the upper half (R8-R15 and S8-S15). To share data between contexts, a program places the data to be shared in one half of either the current processing element's register file or the opposite processing element's register file and activates the alternate register set of the other half. For information on how to activate alternate data registers, see the description of the MODE1 register below.

Each multiplier has a primary or foreground (MRF) register and alternate or background (MRB) results register. A bit in the MODE1 register selects which result register receives the result from the multiplier operation, swapping which register is the current MRF or MRB. This swapping facilitates context switching. Unlike other registers that have alternates, both MRF and MRB are accessible at the same time. Fixed-point multiplies can accumulate results in the MRF or MRB registers, without regard to the state of the MODE1 register. With this arrangement, code can use the result registers as primary and alternate accumulators, or code can use these registers as two parallel accumulators. This feature facilitates complex math.

The MODE1 register controls the access to alternate registers. Table B-2 on page B-5 lists the bits in MODE1. The following bits in the MODE1 register control alternate registers (a 1 enables the alternate set):

- Secondary registers for computational unit results, bit 2 (SRCU)
- Secondary registers for the hi register file, R8–R15 and S8–S15, bit 7 (SRRFH)
- Secondary registers for the lo register file, R0–R7 and S0–S7, bit 10 (SRRFL)

The following example demonstrates how code should handle the one cycle of latency—from the instruction that sets the bit in the MODE1 register until the alternate registers may be accessed. Note that it is possible to use any instruction that does not access the switching register file instead of a NOP instruction.

```
BIT SET MODE1 SRRFL;   /* activate alternate reg. file */
NOP;                   /* wait for access to alternates */
R0 = 7;
```

Multifunction Computations

The processor supports multiple parallel (multifunction) computations by using the parallel data paths within its computational units. These instructions complete in a single cycle, and they combine parallel operation of the multiplier and the ALU or dual ALU functions. The multiple operations perform as if they were in corresponding single function computations. Multifunction computations also handle flags in the same way as the single function computations, except that in the dual add/subtract computation, the ALU flags from the two operations are ORed together.

To work with the available data paths, the computational units constrain which data registers hold the four input operands for multifunction computations. These constraints limit which registers may hold the X input and Y input for the ALU and multiplier.

Figure 2-15 shows a computational unit and indicates which registers may serve as X inputs and Y inputs for the ALU and multiplier. For example, the X input to the ALU can only be `R8`, `R9`, `R10` or `R11`. Note that the shifter is gray in Figure 2-15 to indicate no shifter multifunction operations.

Table 2-12, Table 2-13, Table 2-14, and Table 2-15 list the multifunction computations. For more information on assembly language syntax, see“Instruction Set” in Chapter 8, Instruction Set, and“Computations Reference” in Chapter 9, Computations Reference. Table 2-11 provides the description of the following symbols.

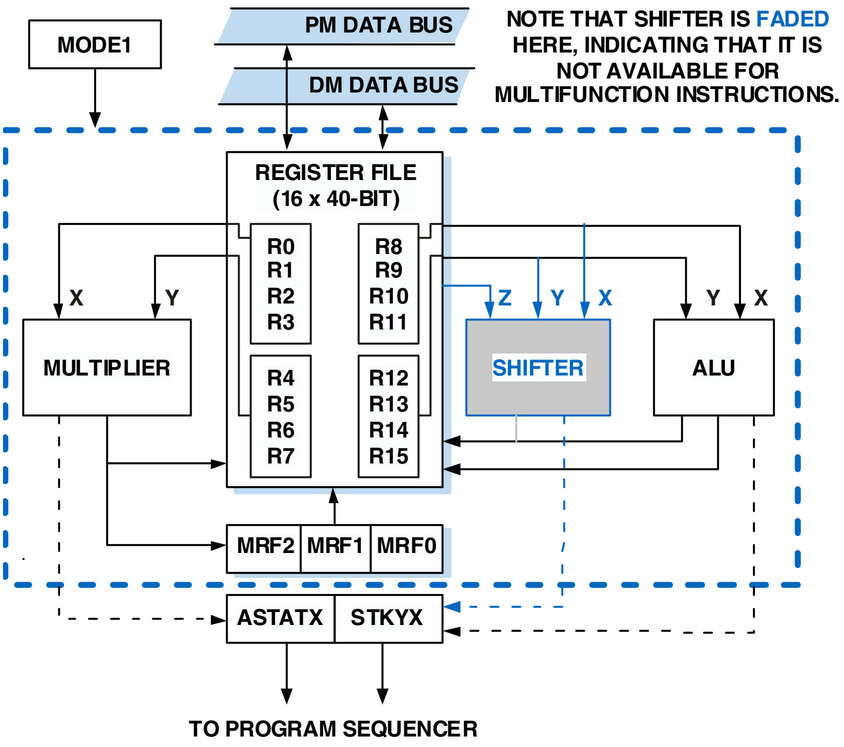


Figure 2-15. Input Registers for Multifunction Computations (ALU and Multiplier)

Table 2-11. Multifunction Computation Symbol Descriptions

Symbol	Description
Rm, Ra, Rs, Rx, Ry	any register file location; fixed-point
Fm, Fa, Fs, Fx, Fy	any register file location; floating-point
R3–0	data file registers R3, R2, R1, or R0
R7-4	data file registers R7, R6, R5 or R4
F3–0	data file registers F3, F2, F1, or F0
F7–4	data file registers F7, F6, F5, or F4
R11–8	data file registers R11, R10, R9, or R8
F11–8	data file registers F11, F10, F9, or F8
R15–12	data file registers R15, R14, R13, or R12
F15–12	data file registers F15, F14, F13, or F12
SSFR	the X input is signed, the Y input is signed, use fractional inputs, and rounded-to-nearest output
SSF	the X input is signed, Y input is signed, use fractional input

Table 2-12. Dual Add and Subtract

Ra = Rx + Ry, Rs = Rx – Ry
Fa = Fx + Fy, Fs = Fx – Fy

Table 2-13. Fixed-Point Multiply and Add, Subtract, or Average

(Any combination of left and right column)		
Rm = R3-0 * R7-4 (SSFR),		Ra = R11-8 + R15-12
MRF = MRF + R3-0 * R7-4 (SSF),		Ra = R11-8 – R15-12
Rm = MRF + R3-0 * R7-4 (SSFR),		Ra = (R11-8 + R15-12)/2
MRF = MRF – R3-0 * R7-4 (SSF),		
Rm = MRF – R3-0 * R7-4 (SSFR),		

Table 2-14. Floating-Point Multiply and ALU Operation

Fm = F3-0 * F7-4, Fa = F11-8 + F15-12
Fm = F3-0 * F7-4, Fa = F11-8 – F15-12
Fm = F3-0 * F7-4, Fa = FLOAT R11-8 by R15-12
Fm = F3-0 * F7-4, Ra = FIX F11-8 by R15-12
Fm = F3-0 * F7-4, Fa = (F11-8 + F15-12)/2
Fm = F3-0 * F7-4, Fa = ABS F11-8
Fm = F3-0 * F7-4, Fa = MAX (F11-8, F15-12)
Fm = F3-0 * F7-4, Fa = MIN (F11-8, F15-12)

Table 2-15. Multiply With Dual Add and Subtract

Rm = R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12, Rs = R11-8 – R15-12
Fm = F3-0 * F7-4, Fa = F11-8 + F15-12, Fs = F11-8 – F15-12

Another type of multifunction operation available on the processor combines transfers between the results and data registers and transfers between memory and data registers. These parallel operations complete in a single cycle. For example, the processor can perform the following multiply and parallel read of data memory:

```
MRF = MRF - R5 * R0, R6 = DM(I1,M2);
```

Or, the processor can perform the following result register transfer and parallel read:

```
R5 = MR1F, R6 = DM(I1,M2);
```

Secondary Processing Element (PEy)

The ADSP-2136x processor contains two sets of computational units and associated register files. As shown in Figure 2-16, these two processing elements (PEx and PEy) support SIMD operation.

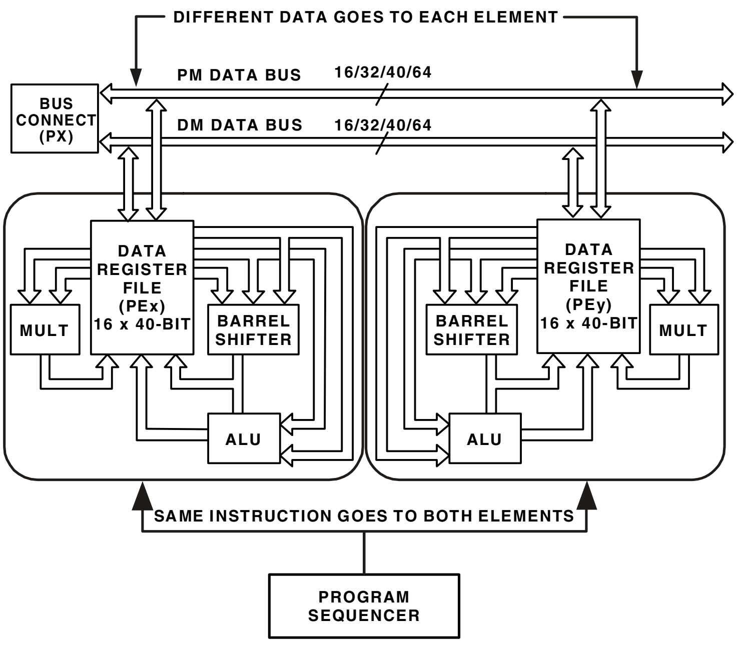

DIFFERENT DATA GOES TO EACH ELEMENT
PM DATA BUS
16/32/40/64
DM DATA BUS
16/32/40/64
BUS CONNECT (PX)
DATA REGISTER FILE (PEx) 16 x 40-BIT
MULT
BARREL SHIFTER
ALU
DATA REGISTER FILE (PEy) 16 x 40-BIT
BARREL SHIFTER
MULT
ALU
SAME INSTRUCTION GOES TO BOTH ELEMENTS
PROGRAM SEQUENCER

Figure 2-16. Block Diagram Showing Secondary Execution Complex

The MODE1 register controls the operating mode of the processing elements. Table B-2 on page B-5 lists the bits in MODE1. The PEYEN bit (bit 21) in the MODE1 register enables or disables the PEy processing element. When PEYEN is cleared (0), the ADSP-2136x processor operates in SISD mode, using only PEx. When the PEYEN bit is set (1), the processor operates in SIMD mode, using the PEx and PEy processing elements. There is a one cycle delay after PEYEN is set or cleared, before the change to or from SIMD mode takes effect.

To support SIMD, the processor performs these parallel operations:

- Dispatches a single instruction to both processing element's computational units
- Loads two sets of data from memory, one for each processing element
- Executes the same instruction simultaneously in both processing elements
- Stores data results from the dual executions to memory

ⓘ Using the information here and in"Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference. It is possible through the SIMD mode's parallelism to double performance over similar algorithms running in SISD (ADSP-2106x processor compatible) mode.

The two processing elements are symmetrical; each contains these functional blocks:

- ALU
- Multiplier primary and alternate result registers
- Shifter
- Data register file and alternate register file

Dual Compute Units Sets

The computational units (ALU, multiplier, and shifter) in PEx and PEy are identical. The data bus connections for the dual computational units permit asymmetric data moves to, from, and between the two processing elements. Identical instructions execute on the PEx and PEy computational units; the difference is the data. The data registers for PEy operations are identified (implicitly) from the PEx registers in the instruction. This implicit relationship between PEx and PEy data registers corresponds to complementary register pairs in Table 2-16. Any universal registers (`Ureg`) that do not appear in Table 2-16 have the same identities in both PEx and PEy. When a computation in SIMD mode refers to a register in the PEx column, the corresponding computation in PEy refers to the complimentary register in the PEy column.

Table 2-16. SIMD Mode Complementary Register Pairs

PEx	PEy	PEx	PEy
R0	S0	R9	S9
R1	S1	R10	S10
R2	S2	R11	S11
R3	S3	R12	S12
R4	S4	R13	S13
R5	S5	R14	S14
R6	S6	ASTATx	ASTATy
R7	S7	STKYx	STKYy
R8	S8		

Table 2-17. Other Complementary Register Pairs

USTAT1	USTAT2
USTAT3	USTAT4
PX1	PX2
MRF	MSF[1]
MRB	MSB[1]

1 These register pairs are not directly accessible by instructions. However, these registers can be read using the multiplier operation MRxF/B = Rn/Rn = MRxF/B. For more information on this instruction, see "Computations Reference" in Chapter 9, Computations Reference.

Dual Register Files

The operand, result busing, and porting are identical in the two 16-entry data register files (one in each PE). The same is true for each 16-entry alternate register files. The transfer direction, data bus, source and destination registers and usage depend on the following conditions:

- **Computational mode:**
  - — Is PEy enabled—`PEYEN` bit=1 in `MODE1` register?
  - — Is the data register file in PEx (`R0`–`R15`, `F0`–`F15`) or PEy (`S0`–`S15`)?
  - — Is the instruction a data register swap between the processing elements?
- **Data addressing mode:**
  - — What is the state of the internal memory data width (`IMDW`) bits in the system control (`SYSCTL`) register?
  - — Is broadcast write enabled— Are `BDCST1,9` bits in `MODE1` register =0?
  - — What is the type of address—long, normal, or short word?
  - — Is long word override (`LW`) specified in the instruction?
  - — What are the states of instruction fields for DAG1 or DAG2?
- **Program sequencing (conditional logic):**
  - — What is the outcome of the instruction's condition comparison on each processing element?

For information on SIMD issues that relate to computational modes, see "SIMD (Computational) Operations" on page 2-49. For information on SIMD issues relating to data addressing, see "Addressing in SISD and SIMD Modes" on page 4-20. For information on SIMD issues relating to program sequencing, see "Summary" on page 3-83.

Dual Alternate Registers

Both register files consist of a primary set of 16 x 40-bit registers and an alternate set of 16 x 40-bit registers. Context switching between the two sets of registers occurs in parallel between the two processing elements. For more information, see "Alternate (Secondary) Data Registers" on page 2-39.

SIMD (Computational) Operations

In SIMD mode, the dual processing elements execute the same instruction, but operate on different data. To support SIMD operation, the elements support a variety of dual data move features.

The processor supports unidirectional and bidirectional register-to-register transfers with the Conditional Compute and Move instruction. All four combinations of inter-register file and intra-register file transfers (PEx ↔ PEx, PEx ↔ PEy, PEy ↔ PEx, and PEy ↔ PEy) are possible in SISD (unidirectional) and SIMD (bidirectional) modes.

In SISD mode (`PEYEN` bit=0), the register-to-register transfers are unidirectional; an operation performed on one processing element is not duplicated on the other processing element. The SISD transfer uses a source register and a destination register. Either register can be in either element's data register file. For a summary of unidirectional transfers, see the upper half of Table 2-18 on page 2-51. In SISD mode a condition for an instruction tests the PEx element only, but it applies to the entire instruction.

In SIMD mode (PEYEN bit=1), register-to-register transfers are bidirectional; an operation performed on one element is duplicated in parallel on the other element. The instruction uses two source registers (one from each element's register file) and two destination registers (one from each element's register file). For a summary of bidirectional transfers, see the lower half of Table 2-18. In SIMD mode, conditional explicit and implicit transfers are tested and executed separately in PEx and PEy, respectively, as detailed in Table 2-18.

Bidirectional register-to-register transfers in SIMD mode are allowed between a data register and a DAG, control, or status register. When the DAG, control, or status register is a source of the transfer, the destination can be a data register. This SIMD transfer duplicates the contents of the source register in a data register in both processing elements.

Careful programming is required when a DAG, control, or status register is a destination of a transfer from a data register. If the destination register has a complement (for example ASTATx and ASTATy), the SIMD transfer moves the contents of the explicit data register into the explicit destination and moves the contents of the implicit data register into the implicit destination (the complement). If the destination register has no complement (for example, I0), only the explicit transfer occurs.

Even if the code uses a conditional operation to select whether the transfer occurs, only the explicit transfer can take place if the destination register has no complement.

When a DAG, control, or status register is both a source and a destination, the data move operation executes the same as if SIMD mode were disabled.

In both SISD and SIMD modes, the processor supports bidirectional register-to-register swaps. The swap occurs between one register in each processing element's data register file.

Registers swaps use the special swap operator, `<->`. A register-to-register swap occurs when registers in different processing elements exchange values; for example `R0 <-> S1`. Only single, 40-bit register-to-register swaps are supported; double register operations are not supported.

When register-to-register swaps are unconditional, they operate the same in SISD mode and SIMD mode. If a condition is added to the instruction in SISD mode, the condition tests only in the PEx element and controls the entire operation. If a condition is added in SIMD mode, the condition tests in both the PEx and PEy elements separately and the halves of the operation are controlled, as detailed in Table 2-18.

Table 2-18. Register-to-Register Move Summary (SISD Versus SIMD)

Mode	Instruction	Explicit Transfer Executed According to PEx	Implicit Transfer Executed According to PEx
SISD[1]	IF condition compute, Rx = Ry;	Rx loaded from Ry	None
	IF condition compute, Rx = Sy;	Rx loaded from Sy	None
	IF condition compute, Sx = Ry;	Sx loaded from Ry	None
	IF condition compute, Sx = Sy;	Sx loaded from Sy	None
	IF condition compute, Rx <-> Sy;	Rx loaded from Sy	Sy loaded from Rx
SIMD[2]	IF condition compute, Rx = Ry;	Rx loaded from Ry	Sx loaded from Sy
	IF condition compute, Rx = Sy;	Rx loaded from Sy	Sx loaded from Ry
	IF condition compute, Sx = Ry;	Sx loaded from Ry	Rx loaded from Sy
	IF condition compute, Sx = Sy;	Sx loaded from Sy	Rx loaded from Ry
	IF condition compute, Rx <-> Sy;[3]	Rx loaded from Sy	Sy loaded from Rx

1 In SISD mode, the conditional applies only to the entire operation and is only tested against PEx's flags. When the condition tests true, the entire operation occurs.

2 In SIMD mode, the conditional applies separately to the explicit and implicit transfers. Where the condition tests true (PEx for the explicit and PEy for the implicit), the operation occurs in that processing element.

3 Register-to- register transfers (R0=S0) and register swaps (R0<->S0) do not cause a PMD bus conflict. These operations use only the DMD bus and a hidden 16-bit bus to perform the two register moves.

SIMD and Status Flags

When the processor is in SIMD mode (PEYEN bit=1), computations on both processing elements generate status flags, producing a logical ORing of the exception status test on each processing element. If one of the four fixed-point or floating-point exceptions is enabled, an exception condition on one or both processing elements generates an exception interrupt. Interrupt service routines (ISRs) must determine which of the processing elements encountered the exception. Returning from a floating-point interrupt does not automatically clear the STKY state. Program code must clear the STKY bits in both processing element's sticky status (STKYx and STKYy) registers as part of the exception service routine. For more information, see "Interrupts and Sequencing" on page 3-68.

3 PROGRAM SEQUENCER

The program sequencer controls program flow by constantly providing the address of the next instruction to be fetched for execution. Program flow in the processor is mostly linear, with the processor executing instructions sequentially. This linear flow varies occasionally when the program branches due to nonsequential program structures, such as those described below. Nonsequential structures direct the processor to execute an instruction that is not at the next sequential address following the current instruction. These structures include:

- **Loops.** One sequence of instructions executes multiple times with zero overhead.
- **Subroutines.** A traditional `CALL` structure where the processor temporarily breaks sequential flow to execute instructions from another part of program memory.
- **Jumps.** Program flow is permanently transferred to another part of program memory.
- **Interrupts.** A runtime event (generally not an instruction) triggers the program sequencer to branch to interrupt-handling subroutines.
- **Idle.** An instruction that causes the core to stop executing further instructions and hold its current state until an interrupt occurs. Then, after the processor services the interrupt, the sequencer resumes normal program execution.

The sequencer uses the blocks shown in Figure 3-1 to execute instructions. The sequencer's address multiplexer selects the value of the next fetch address from several possible sources. The fetched address enters the instruction pipeline which is made up of the fetch registers, decode register, and program counter (PC) register. These registers contain the 24-bit addresses of the instructions currently being fetched, decoded, and executed. The PC register, in conjunction with the PC stack register, stores return addresses and top-of-loop addresses. All addresses generated by the sequencer are 24-bit addresses.

The sequencer handles a series of operations, described in these sections:

- "Instruction Pipeline" on page 3-2
- "Instruction Cache" on page 3-8
- "Branches and Sequencing" on page 3-26
- "Stacks and Sequencing" on page 3-35
- "Loops and Sequencing" on page 3-37
- "Conditional Sequencing" on page 3-40
- "SIMD Mode and Sequencing" on page 3-58
- "Interrupts and Sequencing" on page 3-68

Refer to Figure 3-1 for a description of how each of the functional blocks are related.

Instruction Pipeline

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

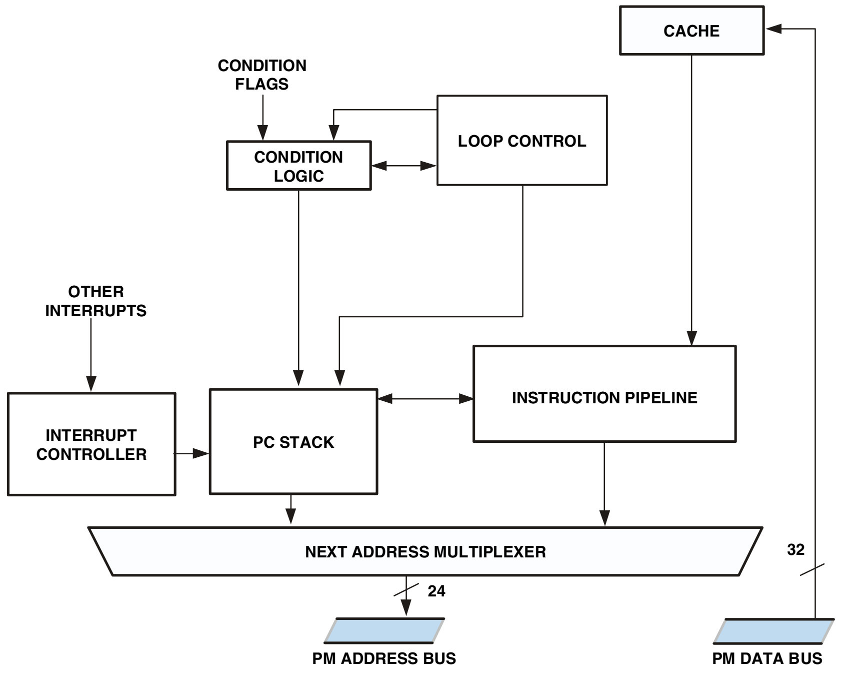


Figure 3-1. Program Sequencer Block Diagram

To achieve a high execution rate while maintaining a simple programming mode, the processor employs a five stage pipeline to process instructions:

1. **Fetch1 stage.** In this stage, the appropriate instruction address is chosen from various sources and driven out to memory.

2. **Fetch2 stage.** This stage is the data phase of the instruction fetch memory access wherein the data address generator (DAG) performs some amount of pre-decode.

3. **Decode stage.** The instruction is decoded and various conditions that control the instruction execution are generated. The main active units in this stage are the DAGs which generate the addresses for various types of functions like data accesses (load/store) and indirect branches.

4. **Address stage.** The addresses generated by the DAGs in the previous stage are driven to the memory through memory interface logic. The addresses for the branch operation are made available to the fetch unit.

5. **Execute stage.** The operations specified in the instruction are executed and the results written back to memory, or the destination registers (for example DAG, universal, system or IOP registers).

In the sequential program flow, when one instruction is being executed, the next four instructions that follow are being processed in the address, decode, fetch2 and fetch1 stages of the instruction pipeline. Sequential program flow usually has a throughput of one instruction per cycle.

Table 3-1 illustrates how the instructions starting at address n are processed by the pipeline. While the instruction at address n is being executed, the instruction n+1 is being processed in the address phase, n+2 in the decode phase, n+3 in the fetch2 phase and n+4 in the fetch1 phase.

Table 3-1. Pipelined Execution Cycles

Cycles	1	2	3	4	5	6	7	8	9
Execute					n	n+1	n+2	n+3	n+4
Address				n	n+1	n+2	n+3	n+4	n+5
Decode			n	n+1	n+2	n+3	n+4	n+5	n+6
Fetch2		n	n+1	n+2	n+3	n+4	n+5	n+6	n+7
Fetch1	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8

While sequential execution takes one core clock cycle per instruction, nonsequential program flow can potentially reduce the instruction throughput. Nonsequential program operations include:

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

Memory Conflicts

Memory conflicts occur when programs attempt to access the certain processor resources simultaneously. A *bus conflict* occurs when an instruction fetch and a data access are made on the same bus. A *block conflict* occurs when multiple accesses are made to the same block in internal memory. The following sections describe these memory conflicts in detail. For additional information, see "Memory and Internal Buses Block Diagram" on page 5-6.

Bus Conflicts

A bus is comprised of two parts, the address bus and the data bus. Because the bus can be accessed simultaneously by different sources (illustrated in Figure 3-1 on page 3-3), there is a potential risk of bus conflicts.

A bus conflict occurs when the PM data bus, normally used to fetch an instruction in each cycle, is used to fetch instruction and to access data. Because of the five stage instruction pipeline, if an instruction at address n uses the PM bus to access data it creates a conflict with the instruction fetch at address n+3, assuming sequential executions.

Note that the cache stores the fetched instruction (n+3), not the instruction requiring the program memory data access.

When the processor first encounters a bus conflict, it must stall for one cycle while the data is transferred, and then fetch the instruction in the following cycle. To prevent the same delay from happening again, the processor automatically writes the fetched instruction to the cache. The sequencer checks the instruction cache on every data access using the PM bus. If the instruction needed is in the cache, a *cache hit* occurs. The instruction fetch from the cache happens in parallel with the program memory data access, without incurring a delay.

If the instruction needed is not in the cache, a *cache miss* occurs, and the instruction fetch (from memory) takes place in the cycle following the program memory data access, incurring one cycle of overhead. The fetched instruction is loaded into the cache (if the cache is enabled and not frozen), so that it is available the next time the same instruction (that requires program memory data) is executed.

Figure 3-2 shows a block diagram of the 2-way set associative instruction cache. The cache holds 32 instruction-address pairs. These pairs (or cache entries) are arranged into 16 (15-0) cache sets according to the four least significant bits (3-0) of their address. The two entries in each set (entry 0 and entry 1) have a valid bit, indicating whether the entry contains a valid instruction. The least recently used (LRU) bit for each set indicates which entry was not placed in the cache last (0=entry 0 and 1=entry 1).

The cache places instructions in entries according to the four LSBs of the instruction's address. When the sequencer checks for an instruction to fetch from the cache, it uses the four address LSBs as an index to a cache set. Within that set, the sequencer checks the addresses of the two entries as it looks for the needed instruction. If the cache contains the instruction, the sequencer uses the entry and updates the `LRU` bit (if necessary) to indicate the entry did not contain the needed instruction.

When the cache does not contain a needed instruction, it loads a new instruction and its address and places them in the least recently used entry of the appropriate cache set. The cache then toggles the `LRU` bit, if necessary.

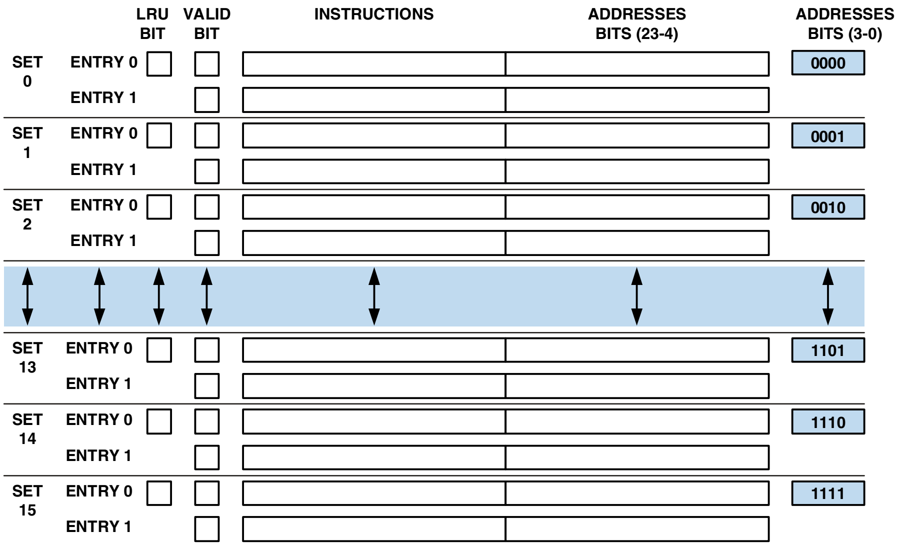


Figure 3-2. Instruction Cache Architecture

Block Conflicts

A block conflict occurs when multiple data accesses are made to the same block in memory from which the instructions are executed. This conflict occurs when accesses to the same block occur in the same cycle by the PM bus, the DM bus, and/or the IOP bus. In the first case, the instruction takes two cycles to complete, with the data being accessed in the first cycle and the instruction in the second. In the latter case, where a dual data access is performed, the processor takes three cycles to complete the instruction.

Block conflicts are not cached.

Instruction Cache

Usually, the sequencer fetches an instruction from memory on each cycle. Occasionally, program memory bus conflicts prevent some of the data and instructions from being fetched in a single cycle, as in the instruction `R0=PM(I8,1);` because it uses the PM bus to fetch data (from the address in `I8`). To alleviate these data flow conflicts, the processor has a 2-way set associative instruction cache that caches instructions that cause these conflicts. This removes the need to fetch the offending instruction from memory and frees both memory blocks and data buses for data accesses. Except for enabling or disabling, the caches operation is completely automatic and transparent, requiring no user intervention. For more information, see "Using the Cache" on page 3-8.

Using the Cache

After a processor reset, the cache is cleared (it contains no instructions), unfrozen, and enabled. From then on, the `MODE2` register controls the operating mode of the instruction cache as shown below.

- **Cache disable.** Bit 4 (`CADIS`) directs the sequencer to disable the cache (if 1) or enable the cache (if 0).
- **Cache freeze.** Bit 19 (`CAFRZ`) directs the sequencer to freeze the contents of the cache (if 1) or let new entries displace the entries in the cache (if 0).

Table B-3 on page B-11 lists the bits in the `MODE2` register.

Freezing the cache prevents any changes to its contents—a cache miss does not result in a new instruction being stored in the cache. Disabling the cache stops its operation completely—all instruction fetches conflicting with program memory data accesses are delayed. These functions are selected by the `CADIS` (cache enable/disable) and `CAFRZ` (cache freeze) bits in the `MODE2` register.

The following restrictions on cache usage should be noted.

- If the cache freeze bit of the MODE2 register is set by instruction *n*, then this feature is effective from the n+2 instruction onwards. This results from the effect latency of the MODE2 register.
- When a program changes the cache mode, an instruction containing a program memory data access must not be placed directly after a cache enable or cache disable instruction. This is because the processor must wait at least one cycle before executing the PM data access. A program should have a NOP (no operation) or other non-conflicting instruction inserted after the cache enable or cache disable instruction.
- The FLUSH CACHE instruction has a latency of one cycle. Using an instruction that contains a PM data access immediately following a FLUSH CACHE instruction is prohibited.

Optimizing Cache Usage

Cache operation is usually efficient and requires no intervention. However, certain ordering in the sequence of instructions can work against the cache's architecture, reducing its efficiency. When the order of PM data accesses and instruction fetches continuously displaces cache entries and loads new entries, the cache does not operate efficiently. Rearranging the order of these instructions remedies this inefficiency. Optionally, a dummy PM read can be inserted to trigger the cache.

When a cache miss occurs, the needed instruction is loaded into the cache so that if the same instruction is needed again, it will be available (that is, a cache hit will occur). However, if another instruction whose address is mapped to the same set displaces this instruction and loads a new instruction, a cache miss occurs. The LRU bits help to reduce the occurrence of a cache miss since at least two other instructions, mapped to the same set, are needed before an instruction is displaced. If three instructions mapped to the same set are all needed repeatedly, cache efficiency (that is, the cache *hit rate*) can go to zero. To keep this from happening, move one or more instructions to a new address that is mapped to a different cache set.

An example of inefficient cache code appears in Table 3-2. The PM bus data access at address 0x101 in the loop, `OUTER`, causes a bus conflict and also causes the cache to load the instruction being fetched at 0x104 (into set 4). Each time the program calls the subroutine, `INNER`, the program memory data accesses at 0x201 and 0x211 displace the instruction at 0x104 by loading the instructions at 0x204 and 0x214 (also into set 4). If the program rarely calls the `INNER` subroutine during the `OUTER` loop execution, the repeated cache loads do not greatly influence performance. If the program frequently calls the subroutine while in the loop, cache inefficiency has a noticeable effect on performance. To improve cache efficiency on this code (if for instance, execution of the `OUTER` instruction of the loop is time critical), rearrange the order of some instructions. Moving the subroutine call up one location (starting at 0x201) also works. By using that order, the two cached instructions end up in cache set 5, instead of set 4.

Table 3-2. Cache Inefficient Code

Address	**Instruction**
0x0100	lcntr = 1024, do Outer until LCE;
0x0101	r0 = dm(i0,m0), pm(i8,m8) = f3;
0x0102	f2 = float r1;
0x0103	f3 = f2 * f2;
0x0104	if eq call (Inner);
0x0105	r1=r0-r15;
0x0106	Outer: f3 = f3 + f4;
0x0107	pm(i8,m8) = f3;
…	
0x0200	Inner: r1 = R13;

Table 3-2. Cache Inefficient Code (Cont'd)

Address	Instruction
0x0201	r14 = pm(i9,m9);
…	
0x0211	pm(i9,m9) = r12;
…	
0x021F	rts;

Instruction Pipeline Stalls

The ADSP-2136x processors use instruction pipeline stalls to ensure correct and efficient program execution. Stalls are used in the following situations.

- "Structural Hazard Stalls" on page 3-12 are incurred when different instructions at various stages of the instruction pipeline attempt to use the same processor resources simultaneously.
- "Data and Control Hazard Stalls" on page 3-14 are incurred when an instruction attempts to read a value from a register or from a condition flag, that has been updated by an earlier instruction, before the value becomes available.
- Stalls are incurred to achieve high performance, when the processor executes a certain sequence of instructions.
- Stalls are incurred to retain effect latency compatible with earlier SHARC processors when the processor executes a certain sequence of instructions.

The following sections describe the various kinds of stalls in detail.

Structural Hazard Stalls

In general, structural stalls occur when different instructions at various stages of the instruction pipeline attempt to use the same resource at the same time during the same cycle. The following sections describe variations of structural stalls and provide examples.

Data Access and Instruction Fetch on the PM Bus

In the instruction `PM(Ip,Mq) = UREG`, the data access over the PM bus by the conflicts with the fetch of instruction n+2 (shown in Table 3-3). In this case the data access completes first. This is true of any program memory data access type instruction. This stall occurs only when the instruction to be fetched is not cached.

Table 3-3. PM Access Conflict

Cycles	1	2	3
Execute		pm(Ip, Mq) = ureg	
Address	pm(Ip, Mq) = ureg		n
Decode	n		n+1
Fetch2	n+1		n+2
Fetch1	n+2	n+2	n+3

1. Cycle1: n+2 Instruction fetch postponed
2. Cycle2: Stall Cycle

Data Access Over the DM and PM Buses

Table 3-4 shows a data access over the DM bus to a particular block of memory and a data access over the PM bus to the same block. These two operations conflict over the single read or write port of the given block. In this example, the data access instruction over the DM bus completes first. The table assumes that the instruction n+2 had previously been cached.

Table 3-4. PM and DM Access Conflicts

Cycles	1	2	3
Execute		dm(I_a, M_b)=R1, pm(I_c, M_d)=R2	dm(I_a, M_b)=R1, pm(I_c, M_d)=R2
Address	dm(I_a, M_b)=R1, pm(I_c, M_d)=R2		n
Decode	n		n+1
Fetch2	n+1		n+2
Fetch1	n+2		n+3
Memory access on dm completes first. **1. Cycle2: Stall Cycle**			

Update and Load Index Register

Updating an I (index) register due to post-modify addressing and loading of a register in the same DAG. The two operations in the same instruction conflicts over the single write port of DAG. The assembler does not allow this type of instruction.

Reading I, M, B, L Registers

Reading an I, B, M or L register in the DAG for pre- or post-modify addressing and reading a register from the same DAG for a store operation. These two operations in the same instruction conflict over the single read port of DAG. The assembler does not allow this type of instruction.

DMA Block Conflict with PM or DM Access

A direct memory access (DMA) by a peripheral such as the parallel port to a particular block of memory and a data/instruction access by the sequencer over the DM or PM bus to the same block of memory. The DMA instruction completes first to ensure that no data overflow or underflow takes place in the processor's peripherals.

Data and Control Hazard Stalls

In general, data and control hazard stalls occur when a register or a condition flag is being updated by an instruction and a subsequent instruction attempts to read the value before the update has actually taken place.

When this occurs, the instruction that is to update the value and the following instruction, (if not dependent on the new value), are allowed to execute. If the following instruction needs the updated value, then that instruction *and* the instructions that follow it in the earlier stages of the instruction pipeline are stalled.

The conditions under which data/control hazard stalls occur are described in the following sections.

Address Generation

Stalls occur when a register in a DAG is loaded and either of the two following instructions (shown in the code examples below) attempts to generate an address based on that register. This is because address generation requires that the value of the related DAG register is read in the decode stage, while any other register load completes in the execution stage of the pipeline. Note that registers can be loaded either by explicit or implicit references (such as in a long word load).

```
M0 = 1;
DM(I2, M0) = R1;   /* stalls for 2 cycles */
```

In the example shown in Table 3-5, `M0` is written back at the end of the execution stage, while the DM access instruction reads `M0` in the decode stage to generate the address. The first instruction is allowed to execute normally, while the remaining instructions are delayed by two cycles.

Also in Table 3-5, the data memory instruction is stalled if the preceding instruction is a load of the `I2`, `B2`, or `L2` registers, regardless of whether circular buffering is enabled or not.

Table 3-5. Indirect Access One Cycle After DAG Register Load

Cycles	1	2	3	4	5
Execute			M0 = 1		
Address		M0 = 1			DM (I2, M0) = R1;
Decode	M0 = 1			DM (I2, M0) = R1;	n
Fetch2	DM (I2, M0) = R1;			n	n+1
Fetch1	n			n+1	n+2
1. Cycle2: Stall cycle					
2. Cycle3: Stall cycle					

In the code example below and Table 3-6, an unrelated instruction is introduced after a write instruction to the DAG. In this case the processor stalls for one cycle only.

```
M0=1;
R0=0x8          /* any unrelated instruction */
Dm(I2,M0)=R1    /* Stalls for one cycle */
```

Table 3-6. Indirect Access Two Cycles After DAG Register Load

Cycles	1	2	3	4	5
Execute		M0 = 1	R0 = 0x8;		DM (I2, M0) = R1;
Address	M0 = 1	R0 = 0x8;		DM (I2, M0) = R1;	n
Decode	R0 = 0x8;		DM (I2, M0) = R1;	n	n+1
Fetch2	DM (I2, M0) = R1;		n	n+1	n+2
Fetch1	n		n+1	n+2	n+3
1. Cycle2: Stall cycle					

Branch

A data stall can also occur when a register in a DAG is loaded and either of the following two instructions shown in the code examples below attempts to generate an indirect target address based on that DAG register for a branch such as a JUMP or CALL. This happens because the address generation requires the values of the related DAG register to be read in the decode stage, while the load of any register completes in the execute stage of the pipeline. The JUMP or CALL itself has three cycles of overhead as described in "Branches and Sequencing" on page 3-26.

```
M8=1;
JUMP (M8,I9);   /* stalls for two cycles */
```

In the example shown in Table 3-7, M8 is written back at the end of the execute stage of the pipeline, while the following JUMP (or CALL) instruction has to read M8 in the decode stage to generate the target address. The first instruction is allowed to complete normally, while all following instructions are stalled for two cycles.

Table 3-7. Indirect Branch One Cycle After DAG Register Load

Cycles	1	2	3	4	5	6	7	8	9
Execute			M8 = 1			jump (M8, I9)	nop	nop	nop
Address		M8 = 1			jump (M8, I9)	nop	nop	nop	j
Decode	M8 = 1			jump (M8, I9)	n→ nop	n+1→ nop	n+2→ nop	j	j+1
Fetch2	jump (M8, I9)			n	n+1	n+2	j	j+1	j+2
Fetch1	n			n+1	n+2	j	j+1	j+2	j+3
J = Branch address **1. Cycle2: Stall cycle** **2. Cycle3: Stall cycle** **3. Cycle4: I9 + M8 computed**									

In the following code example, an unrelated instruction is inserted between the write instruction to the DAG register and the jump instruction requiring address generation. In this instance, the pipeline stalls for only one cycle.

```
M8=1;
R0=0x8;        /* any unrelated instruction */
JUMP (M0,I9);   /* stalls for one cycle */
```

Compute with Post-modify

A control hazard stall occurs when the sequence of three instructions shown below is executed. The first may be a compute instruction, which directly modifies the `ASTATx`, `ASTATy` or `FLAGS` registers, either through an explicit write to the register or through bit manipulation instruction. The second instruction contains a conditional post-modify address generation. The third instruction is either an address generation operation using the same index register or a read of that index register.

The example code and Table 3-8 below shows that when this sequence of instructions is executed, and the third instruction is in the decode stage of the pipeline, the pipeline is stalled for two cycles.

```
R0=PASS R0;          /* ALU instruction, setting a condition
                        flag */
IF EQ DM(I1,M0)=R15  /* conditional post-modify addressing */
DM(I1,0)=R14;        /* address generation using the same I
                        register stalls for two cycles */
```

Table 3-8. Indirect Branch Two Cycles After DAG Register Load

Cycles	1	2	3	4	5	6
Execute		n	n+1			n+2
Address	n	n+1			n+2	n+3
Decode	n+1			n+2	n+3	n+4
Fetch2	n+2			n+3	n+4	n+5
Fetch1	n+3			n+4	n+5	n+6
1. Cycle2: Stall cycle 2. Cycle3: Stall cycle						

When the conditional post-modify instruction is either preceded or followed by instructions other than those involving address generation using the same `I` register, the last instruction stalls the pipeline for one cycle. When the conditional post-modify instruction is either preceded or followed by two or more such unrelated instructions, the pipeline is not stalled.

Note that a conditional instruction based on an ALU generated flag has a dependency on an ALU operation only. This also holds true in the case of multiplier flags and multiplier operations or a `BTF flag` and a `BIT TST` instruction. This is valid for any such kind of dependency.

Also note that when this kind of instruction sequence has other reasons to stall the pipeline, all the stalls arising out of different kinds of dependencies may not merge and some stalls appear as redundant stall cycles.

A JUMP With a LA Modifier Is Used To Abort a Loop

A `JUMP(LA)` stalls the instruction pipeline for one cycle when it is in the address stage of the instruction pipeline.

Loops

A one cycle stall is incurred when a `RTS` (return from subroutine) or `RTI` (return from interrupt) instruction causes the sequencer to return to the last instruction of a loop instruction, and the `RTI`/`RTS` is in the address stage of the instruction pipeline. This is to avoid the coincidence of two implicit operations of the PCSTK—one due to the RTI/RTS instruction and the other due to the possible termination of the loop. The pipeline stalls so that the pop operation from the RTI/RTS is executed first.

Stalls in Conditional Branches

There are three cases related to conditional branches, where the pipeline is stalled for one or more cycles.

1. A control hazard stall occurs when a conditional branch follows a compute or a bit manipulation instruction as shown in the code example and Table 3-9. This occurs because the branch is in the address stage of the pipeline, while the compute and bit manipulation instructions update condition flags at the end of execute phase. (An RTS has three additional overhead cycles. See "Branches and Sequencing" on page 3-26.)

```
R0=R0-1;
If ne RTS;  /* stalls pipe for a cycle */
```

Table 3-9. Conditional Branch Stall

Cycles	1	2	3	4	5	6	7	8
Execute		R0 = R0–1		if ne RTS	nop	nop	nop	r
Address	R0 = R0–1		if ne RTS	nop	nop	nop	r	r+1
Decode	if ne RTS		n→ nop	n+1→ nop	n+2→ nop	r	r+1	r+2
Fetch2	n		n+1	n+2	r	r+1	r+2	r+3
Fetch1	n+1		n+2	r	r+1	r+2	r+3	r+4
r is the instruction branch address **1. Cycle2: Stall cycle** **2. Cycle4: r popped from PC stack**								

2. If the compute involves the multiplier unit and the condition is based on a multiplier flag (as shown in the code sample below), and the conditional branch is in decode stage of the pipeline, the pipeline is stalled for an additional cycle.

   ```
   R0=R0*R1(ssi);
   IF MV CALL (_MultOverFlow); /* stalls for two cycles in
   decode */
   ```

3. The pipeline stalls for two cycles when a branch instruction conditional on `NOT LCE` (loop counter not expired) is in the decode stage and is immediately followed by any instruction involving a change in an `LCE` (loop counter expired) condition, due to the execution of a `DO/UNTIL`, `POP/PUSH`, `JUMP(LA)` or load of the `CURLCNTR` register. A one cycle stall occurs when the instruction is an operation other than a branch.

   Note that if the `CURLCNTR` register changes due to the normal loop-back operation within a counter based loop, the pipeline is not stalled for any branch instruction conditional on the `NOT LCE` condition.

Address Generation Using I Registers After a CJUMP

The following code example shows a two cycle data hazard stall that occurs when DAG1 attempts to generate addresses based on the `I6` register or when either or both of the `I6` or `I7` registers are used as a source of some data transfer operation immediately after a `CJUMP` instruction. This occurs because the `CJUMP` instruction modifies the `I6` register.

Example 1:

```
CJUMP(_SUB1)(DB);  /* executes R2=I6,I6=I7, jump(_sub1) (db) */
DM(I6,M0)=R2;      /*stalls for two cycles */
```

Example 2:

```
CJUMP(_SUB1)(DB);   /* executes R2=I6,I6=I7, jump(_sub1) (db) */
R2=I7;              /* stalls for two cycles */
```

If there is an unrelated instruction before the second instruction, the pipeline stalls for one cycle only. Note that an address generation operation using register `I7` immediately after a `CJUMP` instruction does not stall the pipeline.

Note: `CJUMP` is intended to be used by compiler only. Normally the compiler uses the following sequence of instructions when calling a subroutine, which does not stall the pipeline.

```
CJUMP (_SUB1) (DB);   /* executes R2=I6, I6=I7 */
jump(_sub1)(db)
DM(I7,M0)=R2;         /* stores previous I6  */
DM(I7,M0)=PC;         /* stores return_address-1 */
```

RFRAME Instruction

A data hazard stall occurs when DAG1 attempts to generate addresses based on the `I6` or `I7` registers or when any or both of the `I6` or `I7` registers are used as a source of some data transfer operation immediately after a `RFRAME` instruction. This occurs because `RFRAME` modifies the `I6` and `I7` registers. In this situation, the pipeline is stalled for two cycles.

```
RFRAME;         /* executes I7=I6, I6=dm(0,I7  */
DM(I6,M0)=R2    /* stalls for two cycles  */
```

In a program where there is an unrelated instruction before the `DM` instruction, then the pipeline stalls for one cycle only.

The `RFRAME` instruction is only used by the compiler.

Other Instructions

To achieve high performance, the processor is stalled when it executes one of three specific sequences of instructions. The different conditions under which such cases occur are shown below:

1. When *both* of the operands of the multiplier are produced as a result of either a multiplier or an ALU operation in the immediate preceding instruction, the pipeline is stalled for one cycle as shown in the following example.

   ```
   F0=F0+F4, F1=F0-F4;
   F0=F0*F1;
   /* stalls a cycle since both the operands are produced by
   ALU in the immediately preceding instruction */
   ```

2. When the length of the counter based loop is one, two or four instructions, the pipeline is stalled by one cycle after the `DO/UNTIL` instruction.

3. When a compute operation involving any fixed-point operand register follows a floating point multiply operation, and the instruction involving the fixed-point register is in the decode stage of the pipeline, the pipeline stalls for one cycle as shown in the following example. Note that the actual register used for the operation is not relevant.

   ```
   F0 = F0*F4;
   F5 = FLOAT R1; /* stalls the pipe when in decode */
   ```

Latency

Writes to some of the system registers do not take effect immediately. For example, if a program writes to the `MODE1` register in order to set ALU saturation mode, any ALU operation in the instruction immediately following is not effected. The saturation mode takes effect in the second instruction following the instruction performing the write to `MODE1`. This is referred to as an *effect latency* of one cycle. Also, some registers are not updated on the cycle immediately following a write. It takes an extra cycle before a read of the register returns the updated value. This is referred to as a *read latency* of one cycle.

Note that the effect latency and read latency are counted in a number of processor cycles rather than instruction cycles. Therefore, there may be situations when the effect latency may not be observed, such as when the pipeline stalls or when an interrupt breaks the normal sequence of instructions. Here, the effect latency and the read latency are interpreted as the maximum number of instructions, which is unaffected by the new settings after a write to one register.

Table 3-10 and Table 3-11 summarize the number of extra cycles (latency) for a write to take effect (effect latency) and for a new value to appear in the register (read latency). A 0 (zero) indicates that the write takes effect or appears in the register on the next cycle after the write instruction is executed, and a 1 indicates one extra cycle.

Table 3-10. Sequencer Registers Read and Effect Latencies

Register	Contents	Bits	Read Latency	Effect Latency
FADDR	Fetch address	24	--	--
DADDR	Decode address	24	--	--
PC	Execute address	24	--	--
PCSTK	Top of PC stack	24	0	0
PCSTKP	PC stack pointer	5	1	1
LADDER	Top of loop address stack	32	0	0
CURLCNTR	Top of loop count stack (current loop count)	32	0	0
LCNTR	Loop count for next DO UNTIL loop	32	0	0

Table 3-11. System Registers Read and Effect Latencies

Register	Contents	Bits	Read Latency	Effect Latency
MODE1	Mode control bits	32	0	1
MODE2	Mode control bits	32	0	1
IRPTL	Interrupt latch	32	0	1
IMASK	Interrupt mask	32	0	1
IMASKP	Interrupt mask pointer (for nesting)	32	1	1
MMASK	Mode mask	32	0	1
FLAGS	Flag inputs	32	0	1
LIRPTL	Interrupt latch/mask	32	0	1
ASTATX	Arithmetic status flags	32	0	1
ASTATY	Arithmetic status flags	32	0	1
STKYX	Sticky status flags	32	0	1
STKYY	Sticky status flags	32	0	1
USTAT1	User-defined status flags	32	0	0
USTAT2	User-defined status flags	32	0	0
USTAT3	User-defined status flags	32	0	0
USTAT4	User-defined status flags	32	0	0

The following are examples provide more detail on latency.

- The contents of the `MODE1` and `MODE2` registers are used in the decode stage of the instruction pipeline. To maintain the same effect latency of one cycle, a stall cycle is always added after a write to the `MODE1` or `MODE2` registers. A stall is also introduced when the contents of the `MODE1` and `MODE2` registers are modified through a bit manipulation instruction. The `MODE1` register value also changes when the `PUSH STS` or `POP STS` instructions are executed or when the sequencer branches to, or returns from an ISR (interrupt service routine) which involves a PUSH/POP of the stack. This results in a one cycle stall.

```
MODE1=0x1;       /* enable bit reverse addressing for I8 */
PM(I8,M8)=R14;   /* stalls for a cycle, but unaffected by
                    mode setting */
PM(I8,M8)=R14;   /* performs bit reversed mode of
                    addressing */
```

- When the contents of the ASTAT registers are updated by any operation other than a compute operation, the following instruction stalls for a cycle, if it performs a conditional branch and the condition is anything other than NOT LCE. An example is when ASTAT is explicitly loaded or when the sequencer branches to, or returns from an ISR involving a PUSH/POP of the status stack.

- The effect latency in the case of a FLAGS register is felt when a conditional instruction dependent on the FLAGS register values is executed after modifications to the FLAGS register.

```
BIT SET FLAGS 0x1;   /* set FLAG0  */
IF FLAG0_IN R0=R0+1; /* conditional compute - aborts */
IF FLAG0_IN R0=R0+1;   /* conditional compute - executes */
```

A stall cycle is introduced after a write to the FLAGS register, only if a conditional branch dependent on the FLAGS register settings follows it as the second instruction.

```
BIT SET FLAGS 0x1;     /* set FLAG0  */
IF FLAG0_IN R0=R0+1;   /* unaffected by prior
                          instruction-aborts */
IF FLAG0_IN RTS;     /* stalls a cycle and executes RTS */
```

- A stall cycle results after a write to the ASTATx or ASTATy registers, only if a conditional branch follows it as the second instruction.

```
ASTATX = 0x1;          /* set AZ flag  */
IF NE JUMP(SOMEWHERE);  /* unaffected by prior
                           instruction-aborts */
IF NE RTS;           /* stalls a cycle and executes RTS */
```

- The SYSCTL and BRKCTL registers are memory-mapped registers that serve as control registers. The effect latency for these registers is one cycle and the pipeline is stalled for one cycle following a write to these registers.

Branches and Sequencing

One type of non-sequential program flow that the sequencer supports is branching. A branch occurs when a JUMP or CALL instruction moves execution to a location other than the next sequential address. For descriptions on how to use JUMP and CALL instructions, see "Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference. Briefly, these instructions operate as follows.

- A JUMP or a CALL instruction transfers program flow to another memory location. The difference between a JUMP and a CALL is that a CALL automatically pushes the return address (the next sequential address after the CALL instruction) onto the PC stack. This push makes the address available for the CALL instruction's matching return instruction, (RTS) in the subroutine, allowing an easy return from the subroutine.

- A RETURN instruction causes the sequencer to fetch the instruction at the return address, which is stored at the top of the PC stack. The two types of return instructions are return from subroutine

(RTS) and return from interrupt (RTI). While the RTS pops only the return address off the PC stack, the RTI pops the return address and:

a. Clears the interrupt's bit in the interrupt latch register (IRPTL) and the interrupt mask pointer register (IMASKP). This allows another interrupt to be latched in the IRPTL register and the interrupt mask pointer (IMASKP) register. For more information, see "Interrupt Registers" in the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* and the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

b. Pops the status stack if the ASTATx/y and MODE1 status registers have been pushed for interrupts $\overline{\text{IRQ}}$2-0 or timers.

There are a number of parameters that can be specified for branching instructions:

- JUMP and CALL instructions can be conditional. The program sequencer can evaluate the status conditions to decide whether or not to execute a branch. If no condition is specified, the branch is always taken. For more information on these conditions, see "Interrupts and Sequencing" on page 3-68.

- JUMP and CALL instructions can be immediate or delayed. Because of the instruction pipeline, an immediate branch incurs three lost (overhead) cycles. As shown in Table 3-12 and Table 3-13, the processor aborts the three instructions after the branch, which are in the fetch1, fetch2, and decode stages, while instructions are fetched from the branched address. A delayed branch reduces the overhead to one cycle by allowing the two instructions following the branch to propagate through the instruction pipeline and execute. For more information, see "Delayed Branches" on page 3-29.

- JUMP instructions that appear within a loop or within an interrupt service routine have additional options. For information on the loop abort (LA) option, see "Loops and Sequencing" on page 3-37. For information on the loop reentry (LR) option, see "Restrictions on Ending Loops" on page 3-43. For information on the clear interrupt (CI) option, see "Reusing Interrupts" on page 3-81.

Branches can be direct or indirect. The difference is that with direct branches, the sequencer generates the address while for indirect branches, the PM data address generator (DAG2) produces the address.

Direct branches are JUMP or CALL instructions that use an absolute—not changing at run time—address (such as a program label) or use a PC-relative address. Some instruction examples that cause a direct branch are:

```
CALL fft1024;    /* Where fft1024 is an address label */
JUMP (pc,10);    /* Where (pc,10) is 10-relative addresses after
                  the executing instruction */
```

Indirect branches are JUMP or CALL instructions that use a dynamic address that comes from the PM data address generator (DAG2). For more information on the data address generator, see "Data Address Generators" on page 4-1. Some instruction examples that cause an indirect branch are:

```
JUMP (M8, I12);    /* where (M8, I12) are DAG2 registers */
CALL (M9, I13);    /* where (M9, I13) are DAG2 registers */
```

Conditional Branches

The sequencer supports conditional branches. Conditional branches are JUMP or CALL instructions whose execution is based on testing an IF condition. For more information on condition types in IF condition instructions, see "Conditional Sequencing" on page 3-40. Note that the processor's single-instruction, multiple-data (SIMD) mode influences the execution of conditional branches. For more information, see "Summary" on page 3-83.

Delayed Branches

The instruction pipeline influences how the sequencer handles delayed branches. For immediate branches in which JUMP and CALL instructions are not specified as delayed branches (DB), three instruction cycles are lost (NOP) as the instruction pipeline empties and refills with instructions from the new branch.

As shown in Table 3-12 and Table 3-13, the processor aborts the three instructions after the branch, which are in the fetch1, fetch2 and decode stages. For a CALL instruction, the address of the instruction after the CALL is the return address. During the three lost (no-operation) cycles, the first instruction at the branch address passes through the fetch2, decode and address phases of the instruction pipeline

In the tables that follow, shading indicates aborted instructions, which are followed by NOP instructions.

Table 3-12. Pipelined Execution Cycles for Immediate Branch (Jump/Call)

Cycles	1	2	3	4	5	6	7
Execute	n–2	n–1	n	nop	nop	nop	j
Address	n–1	n	nop	nop	nop	j	j+1
Decode	n	n+1→nop	n+2→nop	n+3→nop	j	j+1	j+2
Fetch2	n+1	n+2	n+3	j	j+1	j+2	j+3
Fetch1	n+2	n+3	j	j+1	j+2	j+3	j+4
n is the branching instruction and j is the instruction branch address **1. Cycle2: n+1 suppressed** **2. Cycle3: n+2 suppressed and for call, n+1 pushed on, to PC stack** **3. Cycle4: n+3 suppressed**							

Table 3-13. Pipelined Execution Cycles for Immediate Branch (Return)

Cycles	1	2	3	4	5	6	7
Execute	n–2	n–1	n	nop	nop	nop	r
Address	n–1	n	nop	nop	nop	r	r+1
Decode	n	n+1→nop	n+2→nop	n+3→nop	r	r+1	r+2
Fetch2	n+1	n+2	n+3	r	r+1	r+2	r+3
Fetch1	n+2	n+3	r	r+1	r+2	r+3	r+4

n is the branching instruction and r is the instruction at the return address
1. Cycle2: n+1 suppressed
2. Cycle3: n+2 suppressed and r popped from PC stack
3. Cycle4: n+3 suppressed

Table 3-14. Pipelined Execution Cycles for Delayed Branch (JUMP or Call)

Cycles	1	2	3	4	5	6	7
Execute	n–2	n–1	n	n+1	n+2	nop	j
Address	n–1	n	n+1	n+2	nop	j	j+1
Decode	n	n+1	n+2	n+3→nop	j	j+1	j+2
Fetch2	n+1	n+2	n+3	j	j+1	j+2	j+3
Fetch1	n+2	n+3	j	j+1	j+2	j+3	j+4

n is the branching instruction and j is the instruction branch address
1. Cycle3: For call n+3 pushed on the PC stack
2. Cycle4: n+3 suppressed

Table 3-15. Pipelined Execution Cycles for Delayed Branch (Return)

Cycles	1	2	3	4	5	6	7
Execute	n–2	n–1	n	n+1	n+2	nop	r
Address	n–1	n	n+1	n+2	nop	r	r+1
Decode	n	n+1	n+2	n+3→nop	r	r+1	r+2
Fetch2	n+1	n+2	n+3	r	r+1	r+2	r+3
Fetch1	n+2	n+3	r	r+1	r+2	r+3	r+4

n is the branching instruction and r is the instruction at the return address
1. Cycle3: r popped from PC stack
2. Cycle4: n+3 suppressed

In JUMP and CALL instructions that use the delayed branch (DB) modifier, one instruction cycle is lost in the instruction pipeline. This is because the processor executes the two instructions after the branch and the third is aborted while the instruction pipeline fills with instructions from the new location. This is shown in the sample code below.

As shown in Table 3-14 and Table 3-15, the processor executes the two instructions after the branch and the third is aborted, while the instruction at the branch address is being processed at the fetch2, decode and address phases of the instruction pipeline. In the case of a CALL instruction, the return address is the third address after the branch instruction. While delayed branches use the instruction pipeline more efficiently than immediate branches, delayed branch code can be harder to implement because of the instructions between the branch instruction and the actual branch. This is described in more detail in "Restrictions and Limitations When Using Delayed Branches" on page 3-32.

Delayed branches and the instruction pipeline also influence interrupt processing. Because the delayed branch instruction and the two instructions that follow it must always be executed sequentially, the processor does not immediately process an interrupt that occurs between a delayed branch instruction and either of the two instructions that follow. Any interrupt that occurs during these instructions is latched, but is not processed until the branch is complete.

This may be useful when two instructions must execute atomically (without interruption), such as when working with semaphores. In the following example, instruction 2 immediately follows instruction 1 in all occasions:

```
jump (pc, 3) (db):
instruction 1;
instruction 2;
```

Note that during a delayed branch, a program can read the PC stack register or PC stack pointer register. This read will show the return address on the PC stack has already been pushed or popped, even though the branch has not occurred yet.

Restrictions and Limitations When Using Delayed Branches

Besides being more challenging to code, delayed branches impose some limitations that stem from the instruction pipeline architecture. Because the delayed branch instruction and the two instructions that follow it must execute sequentially, the instructions in the two locations that follow a delayed branch instruction cannot be any of those described in the following five sections.

Development software for the ADSP-2136x processor should always flag the operations described in the next five sections as code errors in the two locations after a delayed branch instruction.

Normally it is not valid to use two conditional instructions using the (DB) option following each other. But the execution is allowed when these instructions are mutually exclusive:

```
If gt jump (PC, 7) (db);
If le jump (pc, 11) (db);
```

Other Jumps, or Calls With RTI, RTS

These instructions cannot be used when they follow a delayed branch instruction. This is shown in the following code that uses the `JUMP` instruction.

```
jump foo(db);
jump my(db);
r0=r0+r1;
r1=r1+r2;
```

In this case, the delayed branch instruction `r1=r1+r2`, is not executed. Further, the control jumps to `my` instead of `foo`, where the delayed branch instruction is the execution of `foo`.

The exception is for the `JUMP` instruction, which applies for the mutually exclusive conditions EQ (equal), and NE (not equal). If the first EQ condition evaluates true, then the NE conditional jump has no meaning and is the same as a `NOP` instruction. Code samples for these conditions are:

```
if eq jump label1 (db);
if ne jump label1 (db);
nop;
nop;
```

Pushes or Pops of the PC Stack

In this case a push of the PC stack in a delayed branch is followed by a pop. If a value is pushed in the delayed branch of a call, it is first popped in the called subroutine. This is followed by an `RTS` instruction.

```
call foo (db);
push PCSTK;
nop;  /* second push due to PCSTK */
foo;  /* first push because of call */
```

This example shows that when a program pushes the `PCSTK` during a delayed slot, the PC stack pointer is pushed onto the `PCSTK`.

The following instructions are executed prior to executing the `RTS`.

```
pop PCSTK;
RTS (db);
nop;
nop;
```

If pushing the PC stack, a stack pop must be performed first, followed by an `RTS` instruction. If a value is popped inside a delayed branch, whatever subroutine return address is pushed is popped back, which is not allowed.

ⓘ Manipulation of these stacks by using PUSH/POP instructions and explicit writes to these stacks may affect the correct functioning of the loop.

Writes to the PC Stack or PC Stack Pointer

The following two situations may arise when programs attempt to write to the PC stack inside a delayed branch.

1. If programs write into the PC stack inside a jump, one of the following situations can occur.

    a. The PC stack cannot hold a value that has already been pushed onto the PC stack.

       When the PC stack contains a value and a program writes that same value onto the stack, the original value is overwritten by the new value and the original value becomes corrupted.

    b. The PC stack is empty.

       Programs cannot write to the PC stack when they are inside a jump. In this case the PC stack will remain empty.

2. Write to the PC stack inside a call.

   If a program writes to the PC stack inside of a call, the value that is pushed onto the PC stack because of that call is overwritten by the value written onto the PC stack. Therefore, when a program performs an RTS, the program returns to the address pushed onto the PC stack and not to the address pushed while branching to the subroutine. For example:

```
call foo3 (db);
PCSTK=0x9011C;
nop;
```

The value 0x90114 is pushed onto the PC stack, while the value 0x9011C is written to the PC stack. Accordingly, the value 0x90114 is overwritten by the value 0x9011C in the PC stack because values that are pushed onto the stack have precedence over values written to the stack. Therefore, when the program comes back by executing an RTS, the return is to address 0x9011C and not to 0x90114.

IDLE Instruction

An interrupt is needed to come out of the IDLE instruction. If a program places an IDLE instruction inside the delayed branch the processor remains in the idled state because interrupts are latched but not serviced until the program exits a delayed branch.

Stacks and Sequencing

The sequencer includes a program counter (PC) stack, which appears in Figure 3-1 on page 3-3. At the start of a subroutine or loop, the sequencer pushes return addresses for subroutines (CALL instructions with RTI/RTS) and top-of-loop addresses for loops (DO/UNTIL instructions) onto the PC stack. The sequencer pops the PC stack during a return from interrupt (RTI), return from subroutine (RTS), and a loop termination.

The program counter (PC) register is the last stage in the instruction pipeline. It contains the 24-bit address of the instruction the processor executes on the next cycle. The PC register, combined with the PC stack (PCSTK) register, stores return addresses and top-of-loop addresses.

The PC stack is 30 locations deep. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is full. The following bits in the `STKYx` register indicate the PC stack full and empty states.

- **PC stack full.** Bit 21 (`PCFL`) indicates that the PC stack is full (if 1) or not full (if 0)—not a sticky bit, cleared by a `POP`.
- **PC stack empty.** Bit 22 (`PCEM`) indicates that the PC stack is empty (if 1) or not empty (if 0)—not sticky, cleared by a `PUSH`.

Table B-5 on page B-20 lists the bits in the `STKYx` register.

To prevent a PC stack overflow, the PC stack full condition generates the (maskable) stack overflow interrupt (`SOVFI`). This interrupt occurs when the PC stack has 29 of 30 locations filled (the almost full state). The PC stack full interrupt occurs at this point because the PC stack full interrupt service routine needs that last location for its return address.

The address of the top of the PC stack is available in the `PC` stack pointer (`PCSTKP`) register. The value of `PCSTKP` is zero when the PC stack is empty, is 1 through 30 when the stack contains data, and is 31 when the stack overflows. A write to `PCSTKP` takes effect after one cycle of delay. If the PC stack is overflowed, a write to `PCSTKP` has no effect. This register can be read from and written to.

ⓘ Manipulation of these stacks by using `PUSH`/`POP` instructions and explicit writes to these stacks may affect the correct functioning of the loop.

The overflow and full flags provide diagnostic aid only. Programs should not use these flags for runtime recovery from overflow. Note that the status stack, loop stack overflow, and PC stack full conditions trigger a maskable interrupt.

The empty flags can ease stack saves to memory. Programs can monitor the empty flag when saving a stack to memory to determine when the processor has transferred all the values.

Loops and Sequencing

Another type of non-sequential program flow that the sequencer supports is looping. A loop occurs when a DO/UNTIL instruction causes the processor to repeat a sequence of instructions until a condition tests true. Unlike other processors, the ADSP-2136x processors automatically evaluate the loop termination condition and modify the program counter (PC) register appropriately. This allows zero overhead looping.

ⓘ A DO UNTIL instruction may be broadly classified as counter based and arithmetic.

Counter Based Loops

Counter based loops are comprised of instructions that are set to run a specified number of iterations. These iterations are controlled by the loop counter register (LCNTR). The LCNTR register is a non memory-mapped universal register that is initialized to the count value and the loop counter expired (LCE) instruction is used to check the termination condition. Expiration of LCE signals that the loop has completed the number of iterations as per the count value in LCNTR. Loops that terminate with conditions other than LCE have some additional restrictions. For more information, see "Restrictions on Ending Loops" on page 3-43 and "Restrictions on Short Loops" on page 3-46. For more information on condition types in DO/UNTIL instructions, see "Interrupts and Sequencing" on page 3-68.

Note that the processor's SIMD mode influences the execution of loops.

The DO/UNTIL instruction uses the sequencer's loop and condition features, as shown in Figure 3-1 on page 3-3. These features provide efficient software loops without the overhead of additional instructions to branch, test a condition, or decrement a counter. The following code example shows a DO/UNTIL loop that contains three instructions and iterates 30 times.

```
LCNTR = 30, DO the_end UNTIL LCE; /*Loop iterates 30 times*/
R0 = DM(I0,M0), F2 = PM(I8,M8);
R1 = R0-R15;
the_end: F4 = F2 + F3;          /*Last instruction in loop*/
```

When executing a DO/UNTIL instruction, the program sequencer pushes the address of the loop's last instruction and its termination condition onto the loop address stack. The sequencer also pushes the top-of-loop address, (the address of the instruction following the DO/UNTIL instruction), onto the PC stack.

Even though DO/UNTIL loops are executed in the execute stage of the instruction pipeline, the next instruction to be fetched is determined when the DO/UNTIL instruction is in the address stage. This helps to reduce overhead when executing short loops.

The sequencer's instruction pipeline architecture influences loop termination. Because instructions are pipelined, the sequencer must test the termination condition and, if the loop is counter based, decrement the counter before the end of the loop. Based on the test's outcome, the next fetch either exits the loop or returns to the top-of-loop.

The termination condition test occurs when the processor is executing the instruction that is four locations before the last instruction in the loop (at location *e – 4*, where *e* is the end-of-loop address). If the condition tests false, the sequencer repeats the loop and fetches the instruction from the top-of-loop address, which is stored on the top of the PC stack. If the condition tests true, the sequencer terminates the loop and fetches the next instruction after the end of the loop, popping the loop and PC stacks.

A special case of loop termination is the loop abort instruction, `JUMP (LA)`. This instruction causes an automatic loop abort when it occurs inside a loop. When the loop aborts, the sequencer pops the PC and loop address stacks once. If the aborted loop was nested, the single pop of the stacks leaves the correct values in place for the outer loop.

Table 3-16 and Table 3-17 show the instruction pipeline states for loop iteration and termination.

Table 3-16. Pipelined Execution Cycles for Loop Back (Iteration)

Cycles	1	2	3	4	5	6
Execute	e–4	e–3	e–2	e–1	e	b
Address	e–3	e–2	e–1	e	b	b+1
Decode	e–2	e–1	e	b	b+1	b+2
Fetch2	e–1	e	b	b+1	b+2	b+3
Fetch1	e	b	b+1	b+2	b+3	b+4
e is the loop end instruction and b is the loop start instruction **1. Cycle1: Termination condition tests false** **2. Cycle2: Top-of-loop address from PC stack**						

Arithmetic Loops

Arithmetic loops are loops where the termination condition in the `DO/UNTIL` loop is any thing other than `LCE`. An example of arithmetic loop is given below.

```
R8=30;
DO label UNTIL EQ;
R8=R8-1,R0=DM(I0,M0),F2=PM(I8,M8);
R1=R0-R15;
Label: F4=F2+F3
```

Table 3-17. Pipelined Execution Cycles for Loop Termination

Cycles	1	2	3	4	5	6
Execute	e–4	e–3	e–2	e–1	e	e+1
Address	e–3	e–2	e–1	e	e+1	e+2
Decode	e–2	e–1	e	e+1	e+2	e+3
Fetch2	e–1	e	e+1	e+2	e+3	e+4
Fetch1	e	e+1	e+2	e+3	e+4	e+5
e is the loop end instruction 1. Cycle1: Termination condition tests true 2. Cycle2: Loop aborts, PC and loop stacks popped						

Conditional Sequencing

The sequencer supports conditional execution with conditional logic, as illustrated in Figure 3-4 on page 3-84. This logic evaluates conditions for conditional (`IF`) instructions and loop (`DO/UNTIL`) terminations. The conditions are based on information from the arithmetic status registers (`ASTATx` and `ASTATy`), the mode control 1 register (`MODE1`), the flag inputs, and the loop counter. For more information on arithmetic status, see "Using Computational Status" on page 2-15. When in SIMD mode, conditional execution is effected by the arithmetic status of both processing elements. For information on conditional sequencing in SIMD mode, see "Summary" on page 3-83.

Each condition that the processor evaluates has an assembler mnemonic. The condition mnemonics for conditional instructions appear in Table 3-18. For most conditions, the sequencer can test both true and false states. For example, the sequencer can evaluate ALU equal-to-zero (`EQ`) and ALU not-equal-to-zero (`NE`).

To branch conditionally based on the value of a register, a program can use the test flag (`TF`) condition generated from a bit test flag (`BTF`) instruction. The `TF` flag is set or cleared as a result of a `BIT TST` or `BIT XOR` instruction, which can test the contents of any of the processor's system registers, including `STKYx` and `STKYy`.

Table 3-18. IF Condition and DO/UNTIL Termination Mnemonics

Condition From	Description	True If…	Mnemonic
ALU	ALU = 0	AZ = 1	EQ
	ALU ≠ 0	AZ = 0	NE
	ALU > 0	footnote[1]	GT
	ALU < zero	footnote[2]	LT
	ALU ≥ 0	footnote[3]	GE
	ALU ≤ 0	footnote[4]	LE
	ALU carry	AC = 1	AC
	ALU not carry	AC = 0	NOT AC
	ALU overflow	AV = 1	AV
	ALU not overflow	AV = 0	NOT AV
Multiplier	Multiplier overflow	MV = 1	MV
	Multiplier not overflow	MV= 0	NOT MV
	Multiplier sign	MN = 1	MS
	Multiplier not sign	MN = 0	NOT MS
Shifter	Shifter overflow	SV = 1	SV
	Shifter not overflow	SV = 0	NOT SV
	Shifter zero	SZ = 1	SZ
	Shifter not zero	SZ = 0	NOT SZ
Bit Test	Bit test flag true	BTF = 1	TF
	Bit test flag false	BTF = 0	NOT TF

Table 3-18. IF Condition and DO/UNTIL Termination Mnemonics (Cont'd)

Condition From	Description	True If…	Mnemonic
Flag Input	Flag0 asserted	FI0 = 1	FLAG0_IN
	Flag0 not asserted	FI0 = 0	NOT FLAG0_IN
	Flag1 asserted	FI1 = 1	FLAG1_IN
	Flag1 not asserted	FI1 = 0	NOT FLAG1_IN
	Flag2 asserted	FI2 = 1	FLAG2_IN
	Flag2 not asserted	FI2 = 0	NOT FLAG2_IN
	Flag3 asserted	FI3 = 1	FLAG3_IN
	Flag3 not asserted	FI3 = 0	NOT FLAG3_IN
Sequencer	Loop counter expired (Do)	CURLCNTR = 1	LCE
	Loop counter not expired (IF)	CURLCNTR ≠ 1	NOT LCE
	Always false (Do)	Always	FOREVER
	Always true (IF)	Always	TRUE

1 ALU greater than (GT) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN)] or AZ = 0
2 ALU less than (LT) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and ALUSAT)) or (AF and AN and AZ)] = 1
3 ALU greater equal (GE) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN and AZ)] = 0
4 ALU lesser or equal (LT) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN)] or AZ = 1

The two conditions that do not have complements are `LCE/NOT LCE` (loop counter expired/not expired) and `TRUE/FOREVER`. The context of these condition codes determines their interpretation. Programs should use `TRUE` and `NOT LCE` in conditional (`IF`) instructions. Programs should use `FOREVER` and `LCE` to specify loop (`DO/UNTIL`) termination. A `DO FOREVER` instruction executes a loop indefinitely, until an interrupt or reset intervenes.

There are some restrictions on how programs may use conditions in `DO/UNTIL` loops. For more information, see "Restrictions on Ending Loops" on page 3-43 and "Restrictions on Short Loops" on page 3-46.

Restrictions on Ending Loops

The sequencer's loop features (which optimize performance in many ways) limit the types of instructions that may appear at or near the end of the loop. These restrictions include:

- Nested loops cannot use the same end-of-loop instruction address. The sequencer resolves whether to loop back or not, based on the termination condition. If multiple nested loops end on the same instruction, the sequencer exits all the loops when the termination condition for the current loop tests true. There may be other sequencing errors.
- Nested loops with a non-counter based loop as the outer loop must place the end address of the outer loop at least two addresses after the end address of the inner loop.
- Nested loops with a non-counter based loop as the outer loop that use the loop abort instruction, `JUMP (LA)`, to abort the inner loop, may not `JUMP (LA)` to the last instruction of the outer loop.
- For counter based loops, an instruction that writes to the loop counter from memory cannot be used as the fifth-to-last instruction of a counter-based loop (at *e–4*, where *e* is the end-of-loop address).
- An `IF NOT LCE` instruction cannot be used as the instruction that follows a write to `CURLCNTR` from memory.
- Branch (`JUMP` or `CALL`) instructions may not be used as any of the last three instructions of a loop. This *no end-of-loop* branches rule also applies to single instruction, two instruction, and three instruction loops.

  There is one exception to the no end-of-loop branches rule. The last three instructions of a loop may contain an immediate `CALL`, a `CALL` without a `DB` modifier which is paired with a loop re-entry return, or a return (RTS) with the loop reentry modifier (LR). RTS(LR) ensures that the loop counter is not decremented twice. . The immediate CALL may be one of the last three instructions of a loop except for: one instruction, or two instructions, one iteration loops.

- The loop controller uses the loop stack LPSTK, and program control PCSTK for its operation. Manipulation of these stacks by using PUSH/POP instructions and explicit writes to these stacks may affect the correct functioning of the loop.
- The IDLE and EMUIDLE instructions should not be used in:
    1. Counter based loops of one, two or three instructions
    2. The fourth instruction of a counter based loop with four instructions
    3. The fifth from last (e-4) instruction of a loop with more than four instructions
    4. The last three instructions of any arithmetic loop

Note that any modification of the loop resources, such as the PC stack, Loop stack and the CURLCNTR register within the loop may adversely affect the proper functioning of the looping operation and should be avoided.

Short Loops

Short loops are the loops having one, two or three instructions in the body of the loop. Since the body of the loop is less than the depth of the instruction pipeline, short loops give rise to overhead or lost cycles. Some of the overhead is eliminated by handling these short loops in a special way. The following describes how to minimize or eliminate overhead in short loops.

1. Determine the next fetch address at the start of the loop.

   When the `DO/UNTIL` instruction is in the address phase of the instruction pipeline, the next fetch address is determined based on the following rule.

   Assuming `DO/UNTIL` is the *nth* instruction:

   a. Fetch n+1 in the next cycle in the case of one and three instruction loops.

   b. Fetch n+2 in the next cycle in the case of a two-instruction loop.

   c. Fetch the next instruction in all other cases.

2. Special handling

   When a `DO/UNTIL` instruction (n) is in the address stage of the instruction pipeline, the three instructions following it (n+1, n+2, n+3) are also in the pipeline. In the case of a one-instruction loop, the instructions at the fetch2 and fetch1 stages (n+2 and n+3) are not part of the loop body. For two-instruction loops, the instruction at the fetch1 stage (n+3) is not part of the loop body. The unwanted instructions are eliminated by the following.

   a. In the case of one-instruction loop, the instruction (n+1) is held in the decode stage for two additional cycles to allow the instruction pipeline to complete the first fetch from memory.

   b. In the case of two-instruction loop, the processor makes use of an instruction buffer. Whenever a `DO/UNTIL` instruction is detected, the instruction buffer is updated with the instruction following it. The instruction from the instruction buffer (n+1) is substituted for the instruction (n+3), when it moves to the decode stage of the instruction pipeline.

Restrictions on Short Loops

The sequencer's instruction pipeline features (which can optimize performance in many ways) restrict how short loops iterate and terminate. Short loops (one, two, or three instruction loops) terminate in a special way because they are shorter than the instruction pipeline. Counter based loops (`DO`/`UNTIL LCE`) of one, two, or three instructions are not long enough for the sequencer to check the termination condition four instructions before the end of the loop. In these short loops, the sequencer has already looped back when the termination condition is tested. The sequencer provides special handling to prevent overhead (`NOP`) cycles if the loop is iterated a minimum number of times. This is described below.

- A loop that contains one instruction must iterate at least four times (only initial stall).
- A loop that contains two instructions must iterate at least two times (only initial stall).
- A loop that contains three instructions must iterate at least two times.

Short loops that iterate less than minimum number of times, incur up to three cycles of overhead, because there can be up to three aborted instructions after the last iteration to clear the instruction pipeline.

Table 3-19 summarizes all the cases of the loops and the way the termination condition is checked.

Table 3-19. Loop Termination Condition Checks

Body	Iteration	Condition Check[1]	Lost Cycles	Remark
1	1, 2, 3	CURLCNTR==1	3	
1	4 and more	CURLCNTR==4	None	Special case
2	1	CURLCNTR==1	2	

Table 3-19. Loop Termination Condition Checks (Cont'd)

Body	Iteration	Condition Check[1]	Lost Cycles	Remark
2	2 and more	CURLCNTR==2	None	Special case
3	1	CURLCNTR==1	3	
3	2 and more	CURLCNTR==2	None	Special case
4 and more	Any	CURLCNTR==1	None	

1 The termination condition is always checked when the last instruction of the loop is fetched, (when the instruction that is four instructions before the end-of-loop is executed).

Table 3-20 through Table 3-24 show the instruction pipeline execution for counter based single instruction loops. Table 3-25 through Table 3-27 show the pipeline execution for counter based two instruction loops. Table 3-28 and Table 3-29 show the pipeline execution for counter based three instruction loops.

Table 3-20. Pipelined Execution Cycles for Single Instruction Counter Based Loop With Five Iterations

Cycles	1	2	3	4	5	6	7	8
Execute		n	n+1	nop	n+1	n+1	n+1	n+1
Address	n	n+1	nop	n+1	n+1	n+1	n+1	n+2
Decode	n+1	n+1→nop	n+1	n+1	n+1	n+1	n+2	n+3
Fetch2	n+2	n+3	n+3	n+1	n+1	n+2	n+3	n+4
Fetch1	n+3	n+1	n+1	n+1	n+2	n+3	n+4	n+5

n is the loop start instruction and n+2 is the instruction after the loop.

1. Cycle1: Next fetch address determined as n+1. n+1 locked in decode stage.
2. Cycle2: Loop count (LCNTR) equals 5, Decode stalls.
3. Cycle3: n+1 stays in decode, n+1 put into fetch stage.
4. Cycle4: Last instruction fetched, counter expired tests true, n+1 stays in decode.
5. Cycle5: Loop back aborts, PC and Loop stacks popped, the instruction after the loop (n+2) is put in fetch2.
6. Cycle6: Decode stage updates from fetch2.

Table 3-21. Pipelined Execution Cycles for Single Instruction Counter Based Loop With Four Iterations

Cycles	1	2	3	4	5	6	7	8
Execute		n	n+1	nop	n+1	n+1	n+1	n+2
Address	n	n+1	nop	n+1	n+1	n+1	n+2	n+3
Decode	n+1	n+1→nop	n+1	n+1	n+1	n+2	n+3	n+4
Fetch2	n+2	n+3	n+3	n+1	n+2	n+3	n+4	n+5
Fetch1	n+3	n+1	n+1	n+2	n+3	n+4	n+5	n+6

n is the loop start instruction and n+2 is the instruction after the loop.

1. Cycle1: Next fetch address determined as n+1. n+1 locked in decode stage.
2. Cycle2: Loop count (LCNTR) equals 4, decode stalls.
3. Cycle3: LCNTR equals 4, n+1 stays in decode, last instruction fetched, counter expired tests true.
4. Cycle4: n+1 stays in decode, loop back aborts, PC and Loop stacks popped, the next instruction after the loop (n+2) is put into fetch.
5. Cycle5: Decode stage updates from fetch2.

Table 3-22. Pipelined Execution Cycles for Single Instruction Counter Based Loop With Three Iterations

Cycles	1	2	3	4	5	6	7	8	9
Execute		n	n+1	nop	n+1	n+1	nop	nop	nop
Address	n	n+1	nop	n+1	n+1	nop	nop	nop	n+2
Decode	n+1	n+1→nop	n+1	n+1	nop	nop	nop	n+2	n+3
Fetch2	n+2	n+3	n+3	n+1	n+1	n+1	n+2	n+3	n+4
Fetch1	n+3	n+1	n+1	n+1	n+1	n+2	n+3	n+4	n+5

n is the loop start instruction and n+2 is the instruction after the loop.

1. Cycle1: Next fetch address determined as n+1. n+1 locked in decode stage.
2. Cycle2: Loop count (LCNTR) equals 3, decode stalls.
3. Cycle3: n+1 stays in decode, n+1 put in fetch1 stage.
4. Cycle4: n+1 stays in decode, n+1 put in fetch1 stage.
5. Cycle5: Last instruction fetched, counter expired tests true.
6. Cycle6: Loop-back aborts, PC and loop stacks popped, n+2 put in fetch1.

Table 3-23. Pipelined Execution Cycles for Single Instruction Counter Based Loop With Two Iterations

Cycles	1	2	3	4	5	6	7	8	9
Execute		n	n+1	nop	n+1	nop	nop	nop	n+2
Address	n	n+1	nop	n+1	nop	nop	nop	n+2	n+3
Decode	n+1	nop	n+1	nop	nop	nop	n+2	n+3	n+4
Fetch2	n+2	n+3	n+3	n+1	n+1	n+2	n+3	n+4	n+5
Fetch1	n+3	n+1	n+1	n+1	n+2	n+3	n+4	n+5	n+6

n is the loop start instruction and n+2 is the instruction after the loop.

1. Cycle1: Next fetch address determined as n+1. n+1 locked in decode stage 2.
2. Cycle2: Loop count (LCNTR) equals 2, decode stalls.
3. Cycle3: n+1 stays in decode, n+1 put in fetch1 stage.
4. Cycle4: Last instruction fetched, counter expired tests true.
5. Cycle5: Loop-back aborts, PC and loop stacks popped.

Table 3-24. Pipelined Execution Cycles for Single Instruction Counter Based Loop With One Iteration

Cycles	1	2	3	4	5	6	7	8
Execute		n	n+1	nop	nop	nop	nop	n+2
Address	n	n+1	nop	nop	nop	nop	n+2	n+3
Decode	n+1	n+1→nop	n+1→nop	n+1→nop	n+1→nop	n+2	n+3	n+4
Fetch2	n+2	n+3	n+3	n+1	n+2	n+3	n+4	n+5
Fetch1	n+3	n+1	n+1	n+2	n+3	n+4	n+5	n+6

n is the loop start instruction and n+2 is the instruction after the loop.

1. Cycle1: Next fetch address determined as n+1.
2. Cycle2: Loop count (LCNTR) equals 1, decode stalls.
3. Cycle3: Last instruction fetched, counter expired tests true.
4. Cycle5: Loop-back aborts, PC and loop stacks popped, n+2 put in fetch1 stage.

Table 3-25. Pipelined Execution Cycles for Two Instruction Counter Based Loop With Three Iterations

Cycles	1	2	3	4	5	6	7	8	9
Execute		n	n+1	nop	n+2	n+1	n+2	n+1	n+2
Address	n	n+1	nop	n+2	n+1	n+2	n+1	n+2	n+3
Decode	n+1	n+2→nop	n+2	n+1↵	n+2	n+1	n+2	n+3	n+4
Fetch2	n+2	n+3	n+3	n+2	n+1	n+2	n+3	n+4	n+5
Fetch1	n+3	n+2	n+2	n+1	n+2	n+3	n+4	n+5	n+6

Note: n is the loop start instruction and n+3 is the instruction after the loop.
1. Cycle1: Next fetch address determined as n+2.
2. Cycle2: Loop count (LCNTR) equals 3, decode stalls.
3. Cycle3: Next fetch address determined as n+1, n+3 and n+2 held in Fetch2 and Fetch1 respectively.
4. Cycle4: n+1 supplied from instruction buffer into decode, PC stack supplies top of loop address.
5. Cycle5: Last instruction fetched, counter expired tests true.
6. Cycle6: Loop-back aborts, PC and loop stacks popped.

Table 3-26. Pipelined Execution Cycles for Two Instruction Counter Based Loop With Two Iterations

Cycles	1	2	3	4	5	6	7	8
Execute		n	n+1	nop	n+2	n+1	n+2	n+3
Address	n	n+1	nop	n+2	n+1	n+2	n+3	n+4
Decode	n+1	n+2→nop	n+2	n+1↵	n+2	n+3	n+4	n+5
Fetch2	n+2	n+3	n+3	n+2	n+3	n+4	n+5	n+6
Fetch1	n+3	n+2	n+2	n+3	n+4	n+5	n+6	n+7

n is the loop start instruction and n+3 is the instruction after the loop.

1. Cycle1: Next fetch address determined as n+2.
2. Cycle2: Loop count (LCNTR) equals 2, decode stalls.
3. Cycle3: n+3, and n+2 held in fetch2 and fetch1 respectively counter expired tests true.
4. Cycle4: n+1 supplied from instruction buffer into decode, loop-back aborts, PC and loop stacks popped.

Table 3-27. Pipelined Execution Cycles for Two Instruction Counter Based Loop With One Iteration

Cycles	1	2	3	4	5	6	7	8
Execute		n	n+1	nop	n+2	nop	nop	n+3
Address	n	n+1	nop	n+2	nop	nop	n+3	n+4
Decode	n+1	n+2→nop	n+2	n+3→nop	n+2→nop	n+3	n+4	n+5
Fetch2	n+2	n+3	n+3	n+2	n+3	n+4	n+5	n+6
Fetch1	n+3	n+2	n+2	n+3	n+4	n+5	n+6	n+7

n is the loop start instruction and n+3 is the instruction after the loop.

1. Cycle1: Next fetch address determined as n+2.
2. Cycle2: Loop count (LCNTR) equals 1, decode stalls.
3. Cycle3: Last instruction fetched, counter expired tests true.
4. Cycle4: loop-back aborts, PC and loop stacks popped.

Table 3-28. Pipelined Execution Cycles for Three Instruction Counter Based Loop With Two Iterations

Cycles	1	2	3	4	5	6	7	8
Execute		n	n+1	n+2	n+3	n+1	n+2	n+3
Address	n	n+1	n+2	n+3	n+1	n+2	n+3	n+4
Decode	n+1	n+2	n+3	n+1	n+2	n+3	n+4	n+5
Fetch2	n+2	n+3	n+1	n+2	n+3	n+4	n+5	n+6
Fetch1	n+3	n+1	n+2	n+3	n+4	n+5	n+6	n+7

n is the loop start instruction and n+4 is the instruction after the loop.

1. Cycle1: Next fetch address determined as n+1.
2. Cycle2: Loop count (LCNTR) equals 2, fetch address determined by the given rule.
3. Cycle3: Last instruction fetched, counter expired tests true.
4. Cycle4: loop-back aborts, PC and loop stacks popped.

Table 3-29. Pipelined Execution Cycles for Three Instruction Counter Based Loop With One Iteration

Cycles	1	2	3	4	5	6	7	8	9
Execute		n	n+1	n+2	n+3	nop	nop	nop	n+4
Address	n	n+1	n+2	n+3	nop	nop	nop	n+4	n+5
Decode	n+1	n+2	n+3	nop	nop	nop	n+4	n+5	n+6
Fetch2	n+2	n+3	n+1	n+2	n+3	n+4	n+5	n+6	n+7
Fetch1	n+3	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8

n is the loop start instruction and n+4 is the instruction after the loop.

1. Cycle1: Next fetch address determined as n+1.
2. Cycle2: Loop count (LCNTR) equals 1, fetch address determined by the given rule.
3. Cycle4: Last instruction fetched, counter expired tests true.
4. Cycle5: loop-back aborts, PC and loop stacks popped.

Table 3-30. Pipelined Execution Cycles for Four Instruction Counter Based Loop With One Iteration

Cycles	1	2	3	4	5	6	7	8
Execute		n	n+1	nop	n+2	n+3	n+4	n+5
Address	n	n+1	nop	n+2	n+3	n+4	n+5	n+6
Decode	n+1	n+2→nop	n+2	n+3	n+4	n+5	n+6	n+7
Fetch2	n+2	n+3	n+3	n+4	n+5	n+6	n+7	n+8
Fetch1	n+3	n+4	n+4	n+5	n+6	n+7	n+8	n+9

n is the loop start instruction and n+5 is the instruction after the loop
1. Cycle2: Loop count (LCNTR) equals 1, decode stalls
2. Cycle3: Last instruction fetched, Counter expired tests true
3. Cycle4: Loop-back aborts, PC and loop stacks popped

Evaluation of NOT LCE Condition in Counter Based Loops

During the normal execution of the counter based loop, `CURLCNTR` is decremented in every iteration of the loop, when the end-of-loop instruction is fetched. Therefore, the `NOT LCE` condition changes accordingly. Since there are two cycles of latency for the `NOT LCE` condition to change after `CURLCNTR` value has changed, an instruction with a branch on the `NOT LCE` condition also has two cycles of latency. For all other instructions, the latency is one cycle. The following is an example.

```
LCNTR=<COUNT>, DO End UNTIL LCE;
...
Instr(e-4);                /* In last iteration CURLCNTR=1 */
IF NOT LCE CALL (sub1);    /* In all iterations branch is taken */
IF NOT LCE CALL (sub2);    /* In all iterations branch is taken.
                              However, a non-branch instruction
                              aborts only in the last iteration */
IF NOT LCE <any type>;     /* Branch aborts only in the last
                             iteration */
End: Instr(e)
```

Note that the latency is counted in terms of machine cycles and not in terms of instruction cycles. Therefore the behavior is different from that shown in the example, if the pipeline is stalled for some reason (for example for a DMA).

Arithmetic or Non-Counter Based Loops

An arithmetic loop is one in which the loop termination condition is something other than `LCE`. Some restrictions related to non-counter based loops have been mentioned in "Restrictions on Ending Loops" on page 3-43.

In this type of loop, where the body has more than one instruction, the termination condition is checked when the second instruction of the loop body is fetched. In loops that contain a single instruction, the termination condition is checked in every cycle after the `DO`/`UNTIL` instruction is executed.

If the termination condition tests false, then the next instruction is fetched. If the termination condition tests true, then the instruction following the end-of-loop instruction is fetched in the next cycle and the two instructions currently in the fetch1 and fetch2 stages of the instruction pipeline are flushed.

Table 3-31 shows the execution cycles for an arithmetic loop with six instructions.

Table 3-31. Pipelined Execution Cycles for Six Instruction Non-Counter Based Loop

Cycles	1	2	3	4	5	6	7	8	9
Execute	b	b+1	b+2	b+3	b+4	b+5	nop	nop	b+6
Address	b+1	b+2	b+3	b+4	b+5	nop	nop	b+6	b+7
Decode	b+2	b+3	b+4	b+5	b→nop	b+1→nop	b+6	b+7	b+8
Fetch2	b+3	b+4	b+5	b	b+1	b+6	b+7	b+8	b+9
Fetch1	b+4	b+5	b	b+1	b+6	b+7	b+8	b+9	b+10
b is the first instruction of the body of the loop and b+6 is the instruction after the loop **1. Cycle2: Loop back, next fetch instruction is b.** **2. Cycle4: Termination condition tests true, loop-back aborts, PC and loop stacks popped.**									

Short non-counter based loops terminate differently from short counter based loops. These differences stem from the architecture of the pipeline and the conditional logic:

- In a three instruction loop, the termination condition is checked during the cycle where the second instruction is in the fetch1 stage of the pipeline (when the top of the loop is executed). If the condition becomes true, the sequencer completes one full pass (after the current pass) of the loop before exiting.
- In a two instruction loop, the termination condition is checked during the cycle where the last (second from top-of-loop) instruction is in the fetch1 stage of the pipeline. If the condition becomes true when the first instruction is being executed, it tests true during the second instruction as well and one more full pass completes before exiting the loop. If the condition becomes true during the second instruction, two more full passes complete before exiting the loop.

- In a one instruction loop, the sequencer tests the termination condition every cycle. After the cycle when the condition becomes true, the sequencer completes three more iterations of the loop before exiting.

Note that two stages of the pipeline are always flushed in arithmetic loops.

Loop Address Stack

The sequencer's loop support, shown in Figure 3-1 on page 3-3, includes a loop address stack. The loop address stack is six levels deep by 32 bits wide.

The `LADDR` register contains the top entry on the loop address stack. This register is readable and writable over the DM data bus. Reading from and writing to `LADDR` does not move the loop address stack pointer. Only a stack push or pop performed with explicit instructions moves the stack pointer. The `LADDR` register contains the value 0xFFFF FFFF when the loop address stack is empty. A write to `LADDR` has no effect when the loop address stack is empty "Loop Address Stack Register (LADDR)" on page B-32 lists the bits in the `LADDR` register.

The sequencer pushes the termination address, termination code and the loop type information onto the loop address stack when executing a `DO/UNTIL` instruction. The `PUSH LOOP` instruction pushes the stack by changing the pointer only. It does not alter the contents of the loop address stack. Therefore, the `PUSH LOOP` instruction should be usually followed by a write to `LADDR` register.

The stack entry pops off the stack four instructions before the end of its loop's last iteration or on a `POP Loop` instruction. A stack overflow occurs if a seventh entry (one more than full) is pushed onto the loop stack. The stack is empty when no entries are occupied.

The loop stacks' overflow or empty status is available. Because the sequencer keeps the loop stack and loop counter stack synchronized, the same overflow and empty flags apply to both stacks. These flags are in the sticky status register (STKYx). For more information on STKYx, see Table B-5 on page B-20. For more information on how these flags work with the loop stacks, see "Loop Status" on page 3-56. Note that a loop stack overflow causes a maskable interrupt.

Because the sequencer tests the termination condition four instructions before the end of the loop, the loop stack pops before the end of the loop's final iteration. If a program reads LADDR in these last four instructions, the value is already the termination address for the next loop stack entry.

If the loop abort (LA) modifier is specified in the jump, a jump out of the loop pops the loop address stack, PC stack and the loop count stack (if the loop is counter based). This allows the loop to continue to function correctly. However, because only one pop is performed, the loop abort cannot be used to jump more than one level of loop nesting.

Loop Status

The sequencer's loop support, shown in Figure 3-1 on page 3-3, also includes a loop counter stack. The sequencer keeps the loop counter stack synchronized with the loop address stack. Both stacks always have the same number of locations occupied. Because these stacks are synchronized, the same empty and overflow status flags from the STKYx register apply to both stacks.

The loop counter stack is six locations deep by 32 bits wide. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is already full. The following bits in the STKYx register indicate the loop counter stack full and empty states.

- **Loop stacks overflowed.** Bit 25 (LSOV) indicates that the loop counter stack and loop stack are overflowed (if set to 1) or not overflowed (if set to 0)— LSOV is a sticky bit.
- **Loop stacks empty.** Bit 26 (LSEM) indicates that the loop counter stack and loop stack are empty (if set to 1) or not empty (if set to 0)—not sticky, cleared by a PUSH.

Table B-5 on page B-20 lists the bits in the STKYx register.

Within the sequencer, two separate loop counters operate: the current loop counter (CURLCNTR) and loop counter (LCNTR) registers allow access to the loop counter stack. The CURLCNTR register tracks iterations for a loop being executed, and the LCNTR register holds the count value before the loop is executed. The two counters let the processor maintain the count for an outer loop, while a program is setting up the count for an inner loop.

The top entry in the loop counter stack always contains the current loop count. This entry is the CURLCNTR register which is readable and writable over the DM data bus. Reading CURLCNTR when the loop counter stack is empty returns the value 0xFFFF FFFF. A write to CURLCNTR has no effect when the loop counter stack is empty.

The sequencer decrements the value of CURLCNTR for each loop iteration. Because the sequencer tests the termination condition four instruction cycles before the end of the loop, the loop counter also is decremented before the end of the loop. If a program reads CURLCNTR during these last four loop instructions, the value is already the count for the next iteration.

The loop counter stack is popped four instructions before the end of the last loop iteration. When the loop counter stack is popped, the new top entry of the stack becomes the CURLCNTR value—the count in effect for the executing loop. If there is no executing loop, the value of CURLCNTR is 0xFFFF FFFF after the pop.

Writing to CURLCNTR does not cause a stack push. If a program writes a new value to CURLCNTR, the count value of the loop currently executing is affected. When a DO/UNTIL LCE loop is not executing, writing to CURLCNTR has no effect. Because the processor must use CURLCNTR to perform counter based loops, there are some restrictions as to when a program can write to CURLCNTR. See "Restrictions on Ending Loops" on page 3-43 for more information.

The LCNTR register is the next-to-top entry in the loop counter stack. It is the location on the stack that takes effect on the next loop stack push. The LCNTR register is used to set up a count value for a nested loop without changing the count value for the currently executing loop.

A value of zero in LCNTR causes a loop to execute 2^{32} times.

A DO/UNTIL LCE instruction pushes the value of LCNTR onto the loop counter stack, making that value the new CURLCNTR value. Figure 3-3 demonstrates this process for a set of nested loops. The previous CURLCNTR value is preserved one location down in the stack. If a program reads LCNTR when the loop counter stack is full, the stack returns invalid data. When the loop counter stack is full, the stack discards any data written to LCNTR. If a program reads LCNTR during the last four instructions of a terminating loop, the value of LCNTR is the last CURLCNTR value for the loop.

SIMD Mode and Sequencing

The processor supports a SIMD (single-instruction, multiple-data) mode. In this mode, both of the processor's processing elements (PEx and PEy) execute instructions and generate status conditions. For more information on SIMD computations, see "SIMD (Computational) Operations" on page 2-49.

Because the two processing elements can generate different outcomes, the sequencers must evaluate conditions from both elements (in SIMD mode) for conditional (IF) instructions and loop (DO/UNTIL) terminations. The

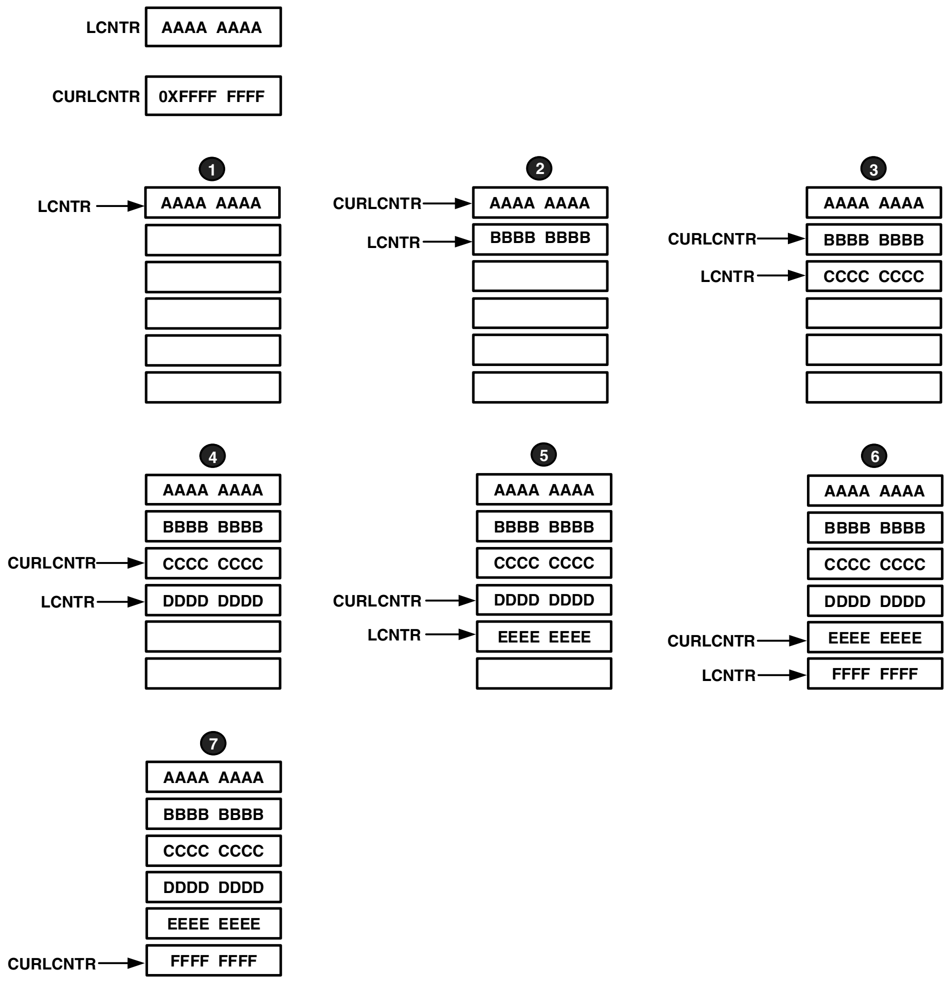


Figure 3-3. Pushing the Loop Counter Stack for Nested Loops

processor records status for the PEx element in the ASTATx and STKYx registers. The processor records status for the PEy element in the ASTATy and STKYy registers. Table B-4 on page B-14 lists the bits in ASTATx and ASTATy, and Table B-5 on page B-20 lists the bits in STKYx and STKYy.

Even though the processor has dual processing elements, the sequencer does not have dual sets of stacks. The sequencer has one PC stack, one loop address stack, and one loop counter stack. The status bits for stacks are in `STKYx` and are not duplicated in `STKYy`. In SIMD mode, the status stack stores both `ASTATx` and `ASTATy` values. A status stack `PUSH` or `POP` instruction in SIMD mode affects both registers in parallel.

While in SIMD mode, the sequencer evaluates conditions from both processing elements for conditional (`IF`) and loop (`DO/UNTIL`) instructions. Table 3-32 on page 3-60 summarizes how the sequencer resolves each conditional test when SIMD mode is enabled.

Table 3-32. Conditional Execution Summary

Conditional Operation	**Conditional Outcome Depends On …**
Compute Operations	Executes in each PE independently depending on condition test in each PE
Branches and Loops	Executes in sequencer depending on ANDing condition test on both PEs
Data Moves (from complementary pair[1] to complementary pair)	Executes move in each PE (and/or memory) independently depending on condition test in each PE
Data Moves (from uncomplementary Ureg register to complementary pair)	Executes move in each PE (and/or memory) independently depending on condition test in each PE; *Ureg* is source for each move
Data Moves (from complementary pair to uncomplementary register[2])	Executes explicit move to uncomplementary universal register depending on the condition test in PEx only; no implicit move occurs
DAG Operations	Executes modify[3] in DAG depending on ORing condition test on both PE's

1 Complementary pairs are registers with SIMD complements, include PEx/y data registers and USTAT1/2, USTAT3/4, ASTATx/y, STKYx/y, and PX1/2 Uregs.

2 Uncomplementary registers are Uregs that do not have SIMD complements.

3 Post-modify operations follow this rule, but pre-modify operations always occur despite the outcome.

Conditional Compute Operations

While in SIMD mode, a conditional compute operation can execute on both processing elements, either element, or neither element, depending on the outcome of the status flag test. Flag testing is independently performed on each processing element.

Conditional Branches and Loops

The processor executes a conditional branch (`JUMP` or `CALL` with `RTI/RTS`) or loop (`DO/UNTIL`) based on the result of ANDing the condition tests on both PEx and PEy. A conditional branch or loop in SIMD mode occurs only when the condition is true in PEx and PEy.

Using complementary conditions (for example `EQ` and `NE`), programs can produce an ORing of the condition tests for branches and loops in SIMD mode. A conditional branch or loop that uses this technique must consist of a series of conditional compute operations. These conditional computes generate `NOP`s on the processing element where a branch or loop does not execute. For more information on programming in SIMD mode, see "Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference.

Conditional Data Moves

The execution of a conditional (`IF`) data move (register-to-register and register-to/from-memory) instruction depends on three factors:

- The explicit data move depends on the evaluation of the conditional test in the PEx processing element.
- The implicit data move depends on the evaluation of the conditional test in the PEy processing element.
- Both moves depend on the types of registers used in the move.

The four cases of SIMD conditional data moves are described in the following sections.

Case #1: Complementary Register Pair Data Move

In this case, data moves from a complementary register pair to a complementary register pair. The processor executes the explicit move depending on the evaluation of the conditional test in the PEx processing element and the implicit move depending on the evaluation of the conditional test in the PEy processing element.

Example 1 – Register-to-Memory Move – PEx Explicit Register

```
IF EQ DM(I0,M0) = R2;
```

For this instruction, the processor is operating in SIMD mode, a register in the PEx data register file is the explicit register, and `I0` is pointing to an even address in internal memory. Indirect addressing is shown in the instructions in the example. However, the same results occur using direct addressing. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-33.

Table 3-33. Register-to-Memory Moves—Complementary Pairs (PEx Explicit Register)

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	No data move occurs	No data move occurs
0	1	No data move occurs from r2 to location I0	s2 transfers to location (I0+1)
1	0	r2 transfers to location I0	No data move occurs from s2 to location (I0+1)
1	1	r2 transfers to location I0	s2 transfers to location (I0+1)

Example 2 Register-to-Memory Move – PEy Explicit Register

```
IF EQ DM(I0,M0) = S2;
```

For this instruction, the processor is operating in SIMD mode, a register in the PEy data register file is the explicit register and `I0` is pointing to an even address in internal memory. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-34.

Table 3-34. Register-to-Memory Moves – Complementary Pairs (PEy Explicit Register)

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	No data move occurs	No data move occurs
0	1	No data move occurs from s2 to location I0	r2 transfers to location I0+1
1	0	s2 transfers to location I0	No data move occurs from r2 to location I0 + 1
1	1	s2 transfers to location I0	r2 transfers to location I0 + 1

Example 3 Register-to-Register Move – PEx Explicit Registers

For the following instructions, the processor is operating in SIMD mode and registers in the PEx data register file are used as the explicit registers. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-35.

```
IF EQ R9 = R2;
IF EQ PX1 = R2;
IF EQ USTAT1 = R2;
```

Table 3-35. Register-to-Register Moves – Complementary Pairs

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	No data move occurs	No data move occurs
0	1	No data move to registers r9, px1, and ustat1 occurs	s2 transfers to registers s9, px2 and ustat2
1	0	r2 transfers to registers r9, px1, and ustat1	No data move to s9, px2, or ustat2 occurs
1	1	r2 transfers to registers r9, px1, and ustat1	s2 transfers to registers s9, px2, and ustat2

Example 4 Register-to-Register Move – PEy Explicit Register

For the following instructions, the processor is operating in SIMD mode and registers in the PEy data register file are used as explicit registers. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-36.

```
IF EQ R9 = S2;
IF EQ PX1 = S2;
IF EQ USTAT1 = S2;
```

Table 3-36. Register-to-Register Moves – Complementary Pairs

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	No data move occurs	No data move occurs
0	1	No data move to registers s9, px and ustat1 occurs	r2 transfers to registers s9, px2, and ustat2

Table 3-36. Register-to-Register Moves – Complementary Pairs (Cont'd)

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
1	0	s2 transfers to registers r9, px1, and ustat1	NO data move to registers s9, px2, and ustat2 occurs
1	1	s2 transfers to registers r9, px1, and ustat1	r2 transfers to registers s9, px2, and ustat2

Case #2: Uncomplimentary-to-Complementary Register Move

In this case, data moves from an uncomplementary register (`Ureg` without a SIMD complement) to a complementary register pair. The processor executes the explicit move depending on the evaluation of the conditional test in the PEx processing element. The processor executes the implicit move depending on the evaluation of the conditional test in the PEy processing element. In each processing element where the move occurs, the content of the source register is duplicated in the destination register.

Note that while `PX1` and `PX2` are complementary registers, the combined `PX` register has no complementary register. For more information, see "Internal Data Bus Exchange" on page 5-7.

For the following instruction the processor is operating in SIMD mode. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-37.

```
IF EQ R1 = PX;
```

Table 3-37. Uncomplimentary-to-Complementary Register Move

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	r1 remains unchanged	s1 remains unchanged
0	1	r1 remains unchanged	s1 gets px value
1	0	r1 gets px value	s1 remains unchanged
1	1	r1 gets px value	s1 gets px value

Case #3: Complementary-to-Uncomplimentary Register Move

In this case data moves from a complementary register pair to an uncomplementary register. The processor executes the explicit move to the uncomplemented universal register, depending on the condition test in the PEx processing element only. The processor does not perform an implicit move.

For all of the following instructions, the processor is operating in SIMD mode. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements for all of the example code samples are shown in Table 3-38.

```
IF EQ PX = R1;
```

Uncomplementary register to DAG move:

```
if EQ m1=PX;
```

DAG to uncomplementary register move:

```
if EQ PX = m1;
```

See also the examples in "Memory" in Chapter 5, Memory, on page 5-12.

Note that `PX1` and `PX2` have compliments, but `PX` as a register is uncomplementary.

DAG to DAG move:

```
if EQ m1 = i15;
```

Complimentary register to DAG move:

```
if EQ i6 = r9;
```

In all the cases described above, the behavior is the same. If the condition in PEx is true, then only the transfer occurs.

Table 3-38. Complementary-to-Uncomplimentary Move

Condition in PEx	Condition in PEy	Result	
AZx	**AZy**	**Explicit**	**Implicit**
0	0	px remains unchanged	no implicit move
0	1	px remains unchanged	no implicit move
1	0	r1 40-bit explicit move to px	no implicit move
1	1	r1 40-bit explicit move to px	no implicit move

For more details on `PX` register transfers, refer to "Internal Data Bus Exchange" on page 5-7.

Case #4: External Memory or IOP Memory Space Data Move

Conditional data moves from a complementary register pair to an uncomplementary register with an access to IOP memory space results in unexpected behavior and should not be used.

Example: Register-to-Memory Moves – IOP Memory Space Data Move

For the following instructions the processor is operating in SIMD mode and the explicit register is either a PEx register or PEy register. `I0` points to IOP memory space. This example shows indirect addressing. However, the same results occur using direct addressing.

```
IF EQ DM(I0,M0) = R2;
IF EQ DM(I0,M0) = S2;
```

Case #5: Uncomplimentary Register Data Move

In the case of memory-to-DAG register moves, the transfer does not occur when both PEx and PEy are false. Otherwise, if either PEx or PEy is true, transfers to the DAG register occur. For example:

```
if EQ m13 = dm(i0,m1);
```

Case #6: Conditional DAG Operations

Conditional post-modify DAG operations update the DAG register based on ORing of the condition tests on both processing elements. Actual data movement involved in a conditional DAG operation is based on independent evaluation of condition tests in PEx and PEy. Only the post-modify update is based on the ORing of the these conditional tests.

Conditional pre-modify DAG operations behave differently. The DAGs always pre-modify an index, independent of the outcome of the condition tests on each processing element.

Interrupts and Sequencing

Interrupts are another type of nonsequential program flow that the sequencer supports. Interrupts may stem from a variety of conditions, both internal and external to the processor. In response to an interrupt, the sequencer processes a subroutine call to a predefined address, called the interrupt vector. The processor assigns a unique vector to each type of interrupt and assigns a priority to each interrupt based on the Interrupt Vector Table (IVT) addressing scheme. For more information, see "Interrupt Registers" in the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* and the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

The processor supports three prioritized, individually-maskable external interrupts, each of which can be programmed to be either level- or edge-triggered. External interrupts occur when an external device asserts one of the processor's interrupt inputs ($\overline{\texttt{IRQ}}$`2-0`). The processor also supports internally generated interrupts. An internal interrupt can occur due to arithmetic exceptions, stack overflows, DMA completion and/or peripheral data buffer status, or circular data buffer overflows. Several factors control the processor's response to an interrupt. When an interrupt occurs, the interrupt is synchronized and latched in the interrupt latch register (`IRPTL`). The processor responds to an interrupt request if:

- The processor is executing instructions or is in an idle state
- The interrupt is not masked
- Interrupts are globally enabled
- A higher priority request is not pending

When the processor responds to an interrupt, the sequencer branches the program execution with a call to the corresponding interrupt vector address. Within the processor's program memory, the interrupt vectors are grouped in an area called the interrupt vector table (IVT). The interrupt vectors in this table are spaced at 4-instruction intervals. Longer service routines can be accommodated by branching to another region of memory. Program execution returns to normal sequencing when return from interrupt (RTI) instruction is executed. Each interrupt vector has associated latch and mask bits.

The ADSP-2136x processor also has extensive programmable interrupt support. These interrupts are described in the hardware references.

To process an interrupt, the processor's program sequencer:

1. Outputs the appropriate interrupt vector address
2. Pushes the current PC value (the return address) onto the PC stack
3. Pushes the current value of the ASTATx/y and MODE1 registers onto the status stack (if the interrupt is $\overline{\text{IRQ}}$2-0 or timer)
4. Resets the appropriate bit in the interrupt latch register (IRPTL and LIRPTL registers)
5. Alters the interrupt mask pointer bits (IMASKP) to reflect the current interrupt nesting state, depending on the nesting mode. The NESTM bit in the MODE1 register determines whether all the interrupts or only the lower priority interrupts are masked during the service routine.

At the end of the interrupt service routine, the sequencer processes the return from interrupt (RTI) instruction and performs the following sequence.

1. Returns to the address stored at the top of the PC stack
2. Pops this value off the PC stack
3. Pops the status stack (if the ASTATx,y and MODE1 status registers were pushed for the $\overline{\text{IRQ2-0}}$, or timer interrupt)
4. Clears the appropriate bit in the interrupt mask pointer (IMASKP)

Between servicing and returning, the sequencer clears the latch bit of the in-progress ISR every cycle until the RTI is executed. When using an ISR, writes into an IOP register (except the serial ports) to clear the interrupt causes some latency. During this delay, the interrupt may be generated a second time. Refer to the "Core Access to IOP Registers" section in the ADSP-21362/3/4/5/6 and ADSP-21367/8/9 hardware references for information on avoiding this.

Except for reset, all interrupt service routines should end with a return-from-interrupt (`RTI`) instruction. After reset, the PC stack is empty, so there is no return address. The last instruction of the reset service routine should be a `JUMP` to the start of the program.

If programs force an interrupt by writing to a bit in the `IRPTL` register, the processor recognizes the interrupt in the following cycle, and four cycles of branching to the interrupt vector follow the recognition cycle.

The processor responds to interrupts in three stages: synchronization (1 cycle), latching and recognition (1 cycle), and branching to the interrupt vector (4 cycles). Table 3-39, Table 3-40, and Table 3-41 show the pipelined execution cycles for interrupt processing.

Table 3-39. Pipelined Execution Cycles for Interrupt Based During Single Cycle Instruction

Cycles	1	2	3	4	5	6	7
Execute	n–2	n–1	nop	nop	nop	nop	v
Address	n–1	n→nop	nop	nop	nop	v	v+1
Decode	n	n+1→nop	n+2→nop	n+3→nop	v	v+1	v+2
Fetch2	n+1	n+2	n+3	v	v+1	v+2	v+3
Fetch1	n+2	n+3	v	v+1	v+2	v+3	v+4

1. Cycle1: Interrupt occurs.
2. Cycle2: Interrupt is latched and recognized, but not processed.
3. Cycle3: n is pushed onto PC stack, fetch of vector address starts.

Table 3-40. Pipelined Execution Cycles for Interrupt During Delayed Branch Instruction

Cycles	1	2	3	4	5	6	7	8	9	10
Execute	n–1	n	n+1	n+2	nop	nop	nop	nop	nop	v
Address	n	n+1	n+2	nop	j→nop	nop	nop	nop	v	v+1
Decode	n+1	n+2	n+3→ nop	j	j+1→ nop	j+2→ nop	j+3→ nop	v	v+1	v+2
Fetch2	n+2	n+3	j	j+1	j+2	j+3	v	v+1	v+2	v+3
Fetch1	n+3	j	j+1	j+2	j+3	v	v+1	v+2	v+3	v+4

N is the delayed branch instruction, J is the jump address, and V is the interrupt vector.

1. Cycle1: Interrupt occurs.
2. Cycle2: Interrupt is latched and recognized, but not processed.
3. Cycle3: N+3 beyond delay slot, interrupt processing delayed.
4. Cycle4: Interrupt processing delayed.
5. Cycle5: Interrupt processed.
6. Cycle6: J pushed onto PC stack, fetch of vector address starts.

Table 3-41. Pipelined Execution Cycles for Interrupt During Instruction With Conflicting PM Data Access (Instruction not Cached)

Cycles	1	2	3	4	5	6	7	8	9
Execute	n–2	n–1	n	nop	nop	nop	nop	nop	v
Address	n–1	n	nop	n+1→ nop	nop	nop	nop	v	v+1
Decode	n	n+1→ nop	n+1	n+2→ nop	n+3→ nop	n+4→ nop	v	v+1	v+2
Fetch2	n+1	n+2	n+2	n+3	n+4	v	v+1	v+2	v+3
Fetch1	n+2	–	n+3	n+4	v	v+1	v+2	v+3	v+4

n is the conflicting instruction, v is the interrupt vector instruction.

1. Cycle1: Interrupt occurs.
2. Cycle2: Interrupt is latched and recognized, but not processed.
3. Cycle3: PM data access stall cycle, n+3 cached interrupt not processed.
4. Cycle4: Interrupt processed.
5. Cycle5: n+1 pushed onto PC stack, fetch of vector address starts.

When an interrupt is caused by the execution of an instruction other than through the direct manipulation of the IRPTL register, the interrupt occurs when the instruction is in the execute stage of the pipeline. The IRPTL register is updated when the sequencer starts fetching the vector address in the following cycle.

For most interrupts, both internal and external, only one instruction is executed after the interrupt occurs (and four instructions are aborted), before the processor fetches and decodes the first instruction of the service routine. Interrupt processing starts two cycles after an arithmetic exception occurs because of the one cycle delay between an arithmetic exception and the STKYx,y register update. There is also a five cycle latency associated with the $\overline{\text{IRQ2-0}}$ interrupts. If an interrupt is latched by explicitly writing into the IRPTL register, then two instructions are executed after that cycle in which IRPTL is written.

If nesting is enabled and a higher priority interrupt occurs immediately after a lower priority interrupt, the service routine of the higher priority interrupt is delayed by one additional cycle. This delay allows the first instruction of the lower priority interrupt routine to be executed before it is interrupted. For more information, see "Nesting Interrupts" on page 3-79.

Certain processor operations that span more than one cycle or which occur at a certain state of the instruction pipeline that involves a change of program flow hold off interrupt processing. If an interrupt occurs during one of these operations, the processor synchronizes and latches the interrupt, but delays its processing. The operations that have delayed interrupt processing are:

- A branch (JUMP or CALL) instruction and the following two cycles, whether they are instructions (in a delayed branch) or a NOP (in a non-delayed branch)

- In addition to the above, the cycle in which a branch is in the address stage of the pipeline along with the last instruction of a counter based loop in the fetch1 stage
- The first of the two cycles used to perform a program memory data access and an instruction fetch (a bus conflict) when the instruction is not cached
- In the case of arithmetic loops, the cycle in which the loop aborts and the following three cycles
- In the case of counter based loops:
    - The cycle in which the counter-expired condition tests true and the following three cycles in the case of loops having less than four instructions in the body
    - The cycle in which the `DO UNTIL LCE` instruction executes and the following cycle for a loop that is composed of one, two or four instructions
- The first four of the five cycles used to fetch and execute the first instruction of an interrupt service routine
- Any cycle in which the core access of internal memory is delayed due to a conflict with the DMA, or the access to the memory-mapped registers is delayed due to wait states

Sensing External Interrupts

For external interrupt pins $\overline{\texttt{IRQ2-0}}$, the processor supports two types of interrupt sensitivity—edge-sensitive and level-sensitive.

The processor detects a level-sensitive interrupt if the signal input is low (active) when sampled on the rising edge of `CLKIN`. A level-sensitive interrupt must go high (inactive) before the processor returns from the interrupt service routine. If a level-sensitive interrupt is still active when the processor samples it after returning from its service routine, the processor treats the signal as a new request. The processor repeats the same interrupt routine without returning to the main program, assuming no higher priority interrupts are active.

The processor detects an edge-sensitive interrupt if the input signal is high (inactive) on one cycle and low (active) on the next cycle when sampled on the rising edge of CLKIN. An edge-sensitive interrupt signal can stay active indefinitely without triggering additional interrupts. To request another interrupt, the signal must go high, then low again.

Edge-sensitive interrupts require less external hardware compared to level-sensitive requests, because negating the request is unnecessary. An advantage of level-sensitive interrupts is that multiple interrupting devices may share a single level-sensitive request line on a wired OR basis, allowing easy system expansion.

The MODE2 register controls external interrupt sensitivity as described below.

- **Interrupt 0 Sensitivity.** Bit 0 ($\overline{\text{IRQ}}$0E) directs the processor to detect $\overline{\text{IRQ}}$0 as edge-sensitive (if 1) or level-sensitive (if 0).
- **Interrupt 1 Sensitivity.** Bit 1 ($\overline{\text{IRQ}}$1E) directs the processor to detect $\overline{\text{IRQ}}$1 as edge-sensitive (if 1) or level-sensitive (if 0).
- **Interrupt 2 Sensitivity.** Bit 2 ($\overline{\text{IRQ}}$2E) directs the processor to detect $\overline{\text{IRQ}}$2 as edge-sensitive (if 1) or level-sensitive (if 0).

Table B-3 on page B-11 lists all of the bits in the MODE2 register.

The processor accepts external interrupts that are asynchronous to the processor's clock (CLKIN), allowing external interrupt signals to change at any time. An external interrupt must be held low at least one CLKIN cycle to guarantee that the processor samples the signal.

External interrupts must meet the setup and hold time requirements relative to the rising edge of CLKIN. For information on interrupt signal timing requirements, see the appropriate ADSP-2136x processor data sheet.

Masking Interrupts

The sequencer supports interrupt masking—latching an interrupt, but not responding to it. Except for the $\overline{\text{RESET}}$ and $\overline{\text{EMU}}$ interrupts, all interrupts are maskable. If a masked interrupt is latched, the processor responds to the latched interrupt if it is later unmasked.

Interrupts can be masked globally or selectively. Bits in the MODE1, IMASK, and LIRPTL registers control interrupt masking as shown in Table B-2 on page B-5 and in the hardware references.

All interrupts are masked at reset except for the non-maskable and boot interrupts. For booting, the processor automatically unmasks and uses the parallel port interrupt (PPI or programmable interrupt 9) or high priority SPI port (default SPIHI or programmable interrupt 1) interrupt after reset. Usage depends on whether the processor is booting from an EPROM, or an SPI master or slave.

Latching Interrupts

When the processor recognizes an interrupt, the processor's interrupt latch (IRPTL and LIRPTL) registers set a bit (latch) to record that the interrupt occurred. The bits set in these registers indicate interrupts that are currently being latched and are pending for execution. Because these registers are readable and writable, any interrupt except reset (RSTI) and emulator (EMUI) can be set or cleared in software.

When an interrupt occurs, the sequencer sets the corresponding bit in IRPTL or LIRPTL register. Throughout the execution of the interrupt's service routine, the processor clears this bit during every cycle. This prevents the same interrupt from being latched while its service routine is executing. After the return from interrupt (RTI), the sequencer stops clearing the latch bit.

If necessary, an interrupt can be reused while it is being serviced. (This is a matter of disabling this automatic clearing of the latch bit.) For more information, see "Reusing Interrupts" on page 3-81.

The interrupt latch bits in IRPTL correspond to interrupt mask bits in the IMASK register. In both registers, the interrupt bits are arranged in their order of priority. The interrupt priority is from 0 (highest) to 31 (lowest). Interrupt priority determines which interrupt must be serviced first, when more than one interrupt occurs in the same cycle. Priority also determines which interrupts are nested when the processor has interrupt nesting enabled. For more information, see "Nesting Interrupts" on page 3-79.

Several events can cause arithmetic interrupts. They are fixed-point overflow (FIXI) and floating-point overflow (FLTOI), underflow (FLTUI), and invalid operation (FLTII). To determine which event caused the interrupt, a program can read the arithmetic status flags in the STKYx or STKYy status registers. Table B-5 on page B-20 lists the bits in these registers. Service routines for arithmetic interrupts must clear the appropriate STKYx or STKYy bits to clear the interrupt. If the bits are not cleared, the interrupt is still active after the return from interrupt (RTI).

Status bits in STKYy apply only in SIMD mode. For more information, see "SIMD (Computational) Operations" on page 2-49.

One event can cause multiple interrupts. The timer decrementing to zero causes two timer expired interrupts to be latched, TMZHI (high priority) and TMZLI (low priority). This feature allows selection of the priority for the timer interrupt. Programs should unmask the timer interrupt with the desired priority and leave the other one masked. If both interrupts are unmasked, the processor services the higher priority interrupt first and then services the lower priority interrupt.

The IRPTL register also provides four software interrupts. When a program sets the latch bit for one of these interrupts (SFT0I, SFT1I, SFT2I, or SFT3I), the sequencer services the interrupt, and the processor branches to the corresponding interrupt routine. Software interrupts have the same behavior as all other maskable interrupts.

Stacking Status During Interrupts

In an interrupt driven system, the processor must be restored to its pre-interrupt state after an interrupt is serviced. The sequencer's status stack eases the return from an interrupt by eliminating some interrupt service overhead like register saves and restores.

The status stack is fifteen locations deep. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is already full. Bits in the STKYx register indicate the status stack full and empty states as describe below.

- **Status stack overflow.** Bit 23 (SSOV) indicates that the status stack is overflowed (if 1) or not overflowed (if 0)—a sticky bit.
- **Status stack empty.** Bit 24 (SSEM) indicates that the status stack is empty (if 1) or not empty (if 0)—not sticky, cleared by a PUSH.

Table B-5 on page B-20 lists all of the bits in the STKYx register.

For some interrupts, ($\overline{\text{IRQ2-0}}$ and timer expired), the sequencer automatically pushes the ASTATx, ASTATy, and MODE1 registers onto the status stack. When the sequencer pushes an entry onto the status stack, the processor uses the MMASK register to clear the corresponding bits in the MODE1 register. All other bit settings remain the same. For more information and an example of how the MMASK and MODE1 registers work together, see "Mode Mask Register (MMASK)" on page B-7.

The sequencer automatically pops the ASTATx, ASTATY, and MODE1 registers from the status stack during the return from interrupt instruction (RTI). In one other case, JUMP (CI), the sequencer pops the stack. For more information, see "Reusing Interrupts" on page 3-81. Only the $\overline{\text{IRQ2-0}}$ and timer expired interrupts cause the sequencer to push an entry onto the status stack. All other interrupts require either explicit saves and restores of effected registers or an explicit push or pop of the stack (PUSH/POP STS).

Pushing the ASTATx, ASTATy, and MODE1 registers preserves the status and control bit settings. This allows a service routine to alter these bits with the knowledge that the original settings are automatically restored upon return from the interrupt.

The top of the status stack contains the current values of ASTATx, ASTATy, and MODE1. Reading and writing these registers does not move the stack pointer. Explicit PUSH or POP instructions do move the status stack pointer.

Nesting Interrupts

The sequencer supports interrupt nesting—responding to another interrupt while a previous interrupt is being serviced. Bits in the MODE1, IMASKP, and LIRPTL registers control interrupt nesting as described below.

- **Interrupt nesting enable.** MODE1 Bit 11 (NESTM). This bit directs the processor to enable (if 1) or disable (if 0) interrupt nesting.
- **Interrupt mask and interrupt mask pointer.** MSKP and IMASKP bits. These bits list the interrupts in priority order and provide a temporary interrupt mask for each nesting level.

Table B-2 on page B-5 lists all of the bits in the MODE1 register. For more information about the IMASKP register and LIRPTL register, see the hardware reference manuals.

When interrupt nesting is enabled, a higher priority interrupt can interrupt a lower priority interrupt's service routine. Lower priority interrupts are latched as they occur, but the processor processes them according to their priority after the nested routines finish.

When interrupt nesting is disabled, a higher priority interrupt cannot interrupt a lower priority interrupt's service routine. Interrupts are latched as they occur and the processor processes them in the order of their occurrence, after the active routine finishes.

Programs should change the interrupt nesting enable (`NESTM`) bit only while outside of an interrupt service routine or during the reset service routine.

If nesting is enabled and a higher priority interrupt occurs immediately after a lower priority interrupt, the service routine of the higher priority interrupt is delayed by one cycle. This delay allows the first instruction of the lower priority interrupt routine to be executed, before it is interrupted.

When servicing nested interrupts, the processor uses the interrupt mask pointer (`IMASKP`) to create a temporary interrupt mask for each level of interrupt nesting but the `IMASK` value is not effected. The processor changes `IMASKP` each time a higher priority interrupt interrupts a lower priority service routine.

The bits in `IMASKP` correspond to the interrupts in their order of priority. When an interrupt occurs, the processor sets its bit in `IMASKP`. If nesting is enabled, the processor uses `IMASKP` to generate a new temporary interrupt mask, masking all interrupts of equal or lower priority to the highest priority bit set in `IMASKP` and keeping higher priority interrupts the same as in `IMASK`. When a return from an interrupt service routine (`RTI`) is executed, the processor clears the highest priority bit set in `IMASKP` and generates a new temporary interrupt mask.

The processor masks all interrupts of equal or lower priority to the highest priority bit set in IMASKP. The bit set in IMASKP that has the highest priority always corresponds to the priority of the interrupt being serviced.

The MSKP bits in the LIRPTL register and the entire set of IMASKP registers are for interrupt controller use only. Modifying these bits interferes with the proper operation of the interrupt controller. Furthermore, explicit bit manipulation of any of the bits in the LIRPTL register, while IRPTEN (bit 12 in the MODE1 register) is set, causes an interrupt to be serviced twice.

Reusing Interrupts

When an interrupt occurs, the sequencer sets the corresponding bit in the IRPTL register. During execution of the service routine, the sequencer keeps this bit cleared which prevents the same interrupt from being latched while its service routine is already executing. If necessary, programs may reuse an interrupt while it is being serviced. Using a jump clear interrupt instruction, (JUMP (CI)) in the interrupt service routine clears the interrupt, allowing its reuse while the service routine is executing.

The JUMP (CI) instruction reduces an interrupt service routine to a normal subroutine, clearing the appropriate bit in the interrupt latch and interrupt mask pointer and popping the status stack. After the JUMP (CI) instruction, the processor stops automatically clearing the interrupt's latch bit, allowing the interrupt to latch again.

When returning from a subroutine entered with a JUMP (CI) instruction, a program must use a return loop reentry instruction RTS (LR), instead of an RTI instruction. For more information, see "Restrictions on Ending Loops" on page 3-43. The following example shows an interrupt service routine that is reduced to a subroutine with the (CI) modifier.

```
instr1; /*Interrupt entry from main program*/

JUMP(PC,4) (DB,CI); /*Clear interrupt status*/
```

```
instr3;

instr4;

instr5;

instr6;

RTS (LR); /*Use LR modifier with return from subroutine*/
```

The `JUMP (PC,4)(DB,CI)` instruction only continues linear execution flow by jumping to the location `PC + 4 (instr6)`. The two intervening instructions `(instr3, instr4)` are executed and `instr5` aborted because of the delayed branch `(DB)`. This `JUMP` instruction is only an example—a `JUMP (CI)` can perform a `JUMP` to any location.

Interrupting IDLE

The sequencer supports placing the processor in `IDLE`—a special instruction that halts the processor core in a low power state. The processor is in the halt state until an external interrupt, timer interrupt, or DMA interrupt occurs and the ISR executed. When executing an `IDLE` instruction, the sequencer fetches one more instruction at the current fetch address and then suspends the operation. The processor's I/O processor is not affected by the `IDLE` instruction—DMA transfers to or from internal memory continue uninterrupted. The processor's internal clock and timer (if enabled) continue to run during `IDLE`. When an external interrupt, or timer interrupt occurs, the processor responds normally. After a five cycle latency to fetch the first instruction of the interrupt service routine, the processor continues to execute the instructions normally.

Summary

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

With selective caching, the instruction cache lets the processor access data in program memory and fetch an instruction (from the cache) in the same cycle. The DAG2 data address generator outputs program memory data addresses.

The sequencer evaluates conditional instructions and loop termination conditions by using information from the status registers. The loop address stack and loop counter stack support nested loops. The status stack stores status registers for implementing nested interrupt routines.

Figure 3-4 identifies all the functional blocks and their relationship to one another in detail.

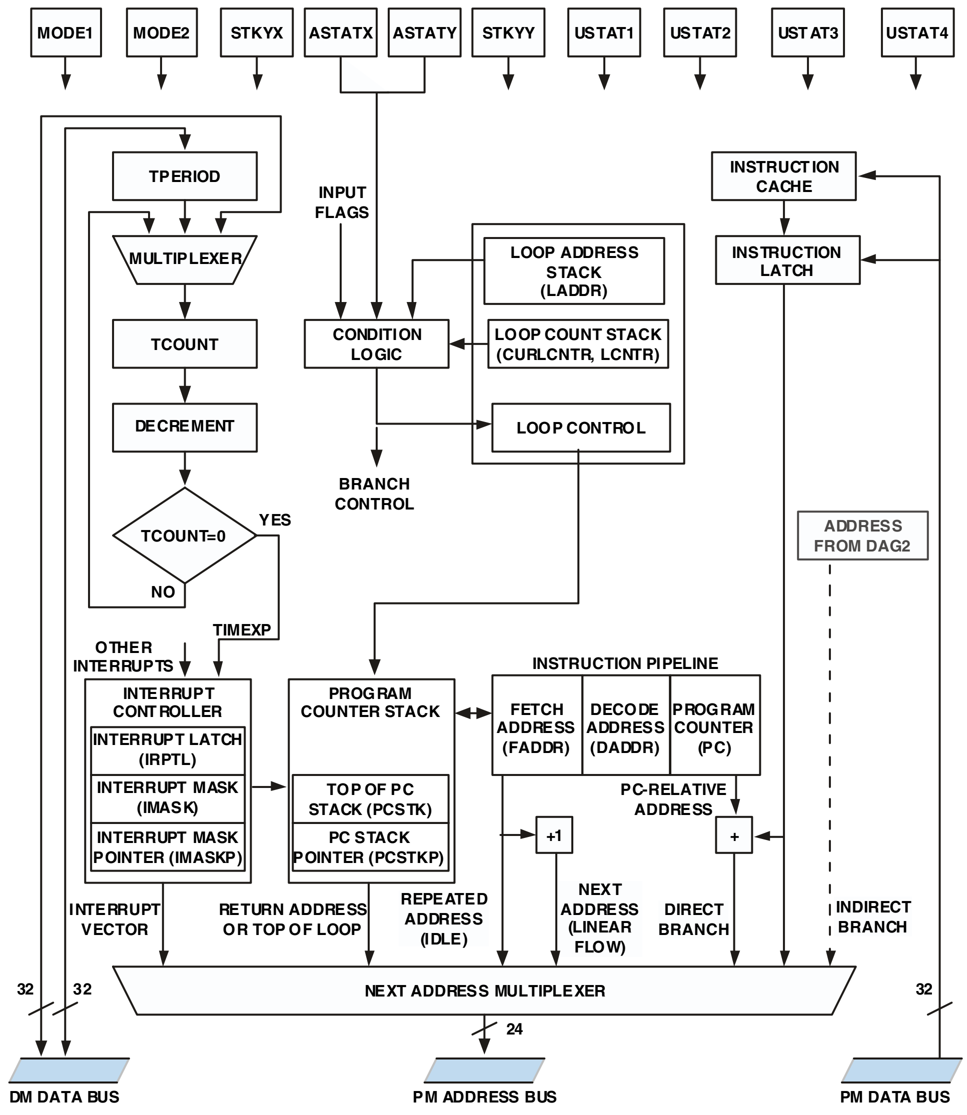


Figure 3-4. Program Sequencer Block Diagram

Table 3-10 on page 3-23 and Table 3-11 on page 3-24 list the registers within and related to the program sequencer. All registers in the program sequencer are universal registers (`Uregs`), so they are accessible to other universal registers and to data memory. All of the sequencer's registers and the top of stacks are readable and writable, except for the fetch, decode, and PC. Pushing or popping the PC stack is done with a write to the PC stack pointer, which is readable and writable. Pushing or popping the loop address stack requires explicit instructions.

A set of system control registers configures or provides input to the sequencer. These registers appear across the top and within the interrupt controller and are shown in Figure 3-1 on page 3-3. A bit manipulation instruction permits setting, clearing, toggling, or testing specific bits in the system registers. For information on this instruction (bit) and the ADSP-2136x Instruction Set, see "Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference. Writes to some of these registers do not take effect on the next cycle. For example, after a write to the `MODE1` register enables ALU saturation mode, the change takes effect two cycles after the write. Also, some of these registers do not update on the cycle immediately following a write. An extra cycle is required before a register read returns the new value.

4 DATA ADDRESS GENERATORS

The processor's data address generators (DAGs) generate addresses for data moves to and from data memory (DM) and program memory (PM). By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address. The DAGs architecture, which appears in Figure 4-1, supports several functions that minimize overhead in data access routines. These functions include:

- **Supply address and post-modify.** Provides an address during a data move and auto-increments the stored address for the next move.
- **Supply pre-modified address.** Provides a modified address during a data move without incrementing the stored address.
- **Modify address.** Increments the stored address without performing a data move.
- **Bit-reverse address.** Provides a bit-reversed address during a data move without reversing the stored address, as well as an instruction to explicitly bit-reverse the supplied address.
- **Broadcast data moves.** Performs dual data moves to complementary registers in each processing element to support single-instruction multiple-data (SIMD) mode.
- **Circular Buffering.** Supports addressing a data buffer with predefined boundaries, wrapping around to cycle through this buffer repeatedly in a circular pattern.

As shown in Figure 4-1, each DAG has four types of registers. These registers hold the values that the DAG uses for generating addresses. The four types of registers are:

- **Index registers (I0–I7 for DAG1 and I8–I15 for DAG2).** An index register holds an address and acts as a pointer to memory. For example, the DAG interprets `DM(I0,0)` and `PM(I8,0)` syntax in an instruction as addresses.
- **Modify registers (M0–M7 for DAG1 and M8–M15 for DAG2).** A modify register provides the increment or step size by which an index register is pre- or post-modified during a register move. For example, the `DM(I0,M1)` instruction directs the DAG to output the address in register `I0` then modify the contents of `I0` using the `M1` register.
- **Length and base registers (L0–L7 and B0–B7 for DAG1 and L8–L15 and B8–B15 for DAG2).** Length and base registers set the range of addresses and the starting address for a circular buffer. For more information on circular buffers, see "Addressing Circular Buffers" on page 4-13.

Setting DAG Modes

The `MODE1` register controls the operating mode of the DAGs as described below.

- **Circular buffering enable.** Bit 24 (`CBUFEN`) enables (if 1) or disables (if 0) circular buffering.
- **Broadcast register loading enable, DAG1-I1.** Bit 23 (`BDCST1`) enables register broadcast loads to complementary registers from `I1` indexed moves (if 1) or disables broadcast loads (if 0).

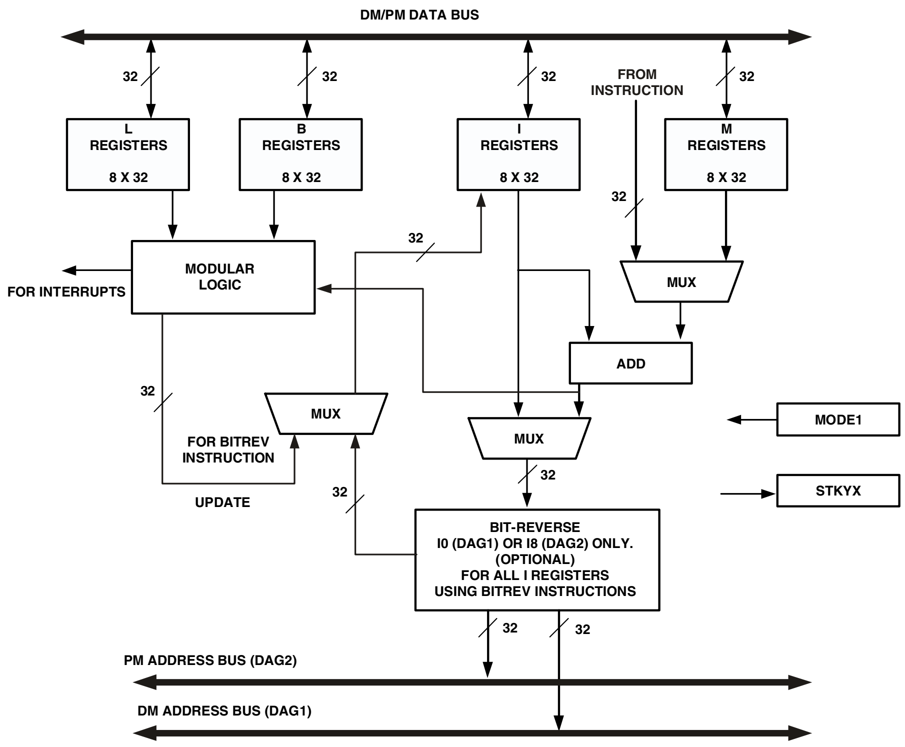


Figure 4-1. Data Address Generator (DAG) Block Diagram

- **Broadcast register loading enable, DAG2–I9.** Bit 22 (`BDCST9`) enables register broadcast loads to complementary registers from `I9` indexed moves (if 1) or disables broadcast loads (if 0).

- **SIMD mode enable.** Bit 21 (`PEYEN`) enables computations in PEy—SIMD mode—(if 1) or disables PEy—SISD mode—(if 0). For more information on SIMD mode, see "SIMD (Computational) Operations" on page 2-49.

- **Secondary registers for DAG2 lo, I, M, L, B8-11.** Bit 6 (`SRD2L`)
  Secondary registers for DAG2 hi, I, M, L, B12–15. Bit 5 (`SRD2H`)
  Secondary registers for DAG1 lo, I, M, L, B0–3. Bit 4 (`SRD1L`)

Secondary registers for DAG1 hi, I, M, L, B4–7. Bit 3 (`SRD1H`) These bits select the corresponding secondary register set (if 1) or select the corresponding primary register set—the set that is available at reset—(if 0).

- **Bit-reverse addressing enable, DAG1–I0.** Bit 1 (`BR0`) enables bit-reversed addressing on `I0` indexed moves (if 1) or disables bit-reversed addressing (if 0).
- **Bit-reverse addressing enable, DAG2–I8.** Bit 0 (`BR8`) enables bit-reversed addressing on `I8` indexed moves (if 1) or disables bit-reversed addressing (if 0).

Table B-2 on page B-5 lists all of the bits in the `MODE1`register.

Circular Buffering Mode

The `CBUFEN` bit in the `MODE1` register enables circular buffering—a mode where the DAG supplies addresses that range within a constrained buffer length (set with an `L` register). Circular buffers start at a base address (set with a `B` register), and increment addresses on each access by a modify value (set with an `M` register).

The circular buffer enable bit (`CBUFEN`) in the `MODE1` register is cleared (= 0) upon reset.

On previous SHARC processors (ADSP-21060/1/2 and ADSP-21065L), circular buffering is always enabled. For code compatibility, programs ported to the ADSP-2136x processors should include the instruction:

```
Bit Set Mode1 CBUFEN;
```

For more information on setting up and using circular buffers, see "Addressing Circular Buffers" on page 4-13. When using circular buffers, the DAGs can generate an interrupt on buffer overflow (wraparound). For more information, see "Using DAG Status" on page 4-9.

Broadcast Loading Mode

The `BDCST1` and `BDCST9` bits in the `MODE1` register enable broadcast loading. An example of broadcast loading is when a program uses one load command to load multiple registers. When the `BDCST1` bit is set (=1), the DAG performs a dual data register load on instructions that use the `I1` register for the address. The DAG loads both the named register (explicit register) in one processing element and loads that register's complementary register (implicit register) in the other processing element. The `BDCST9` bit in the `MODE1` register enables this feature for the `I9` register.

Enabling either DAG register to perform a broadcast load has no effect on register stores or loads to universal registers (*`Uregs`*). The one exception is the register file data registers. Table 4-1 demonstrates the effects of a register load operation on both processing elements with register load broadcasting enabled. In Table 4-1, note that `Rx` and `Sx` are complementary data registers.

Table 4-1. Dual Processing Element Register Load Broadcasts

Instruction syntax	Rx = DM(I1,Ma); {Syntax #1}
	Rx = PM(I9,Mb); {Syntax #2}
	Rx = DM(I1,Ma), Rx = PM(I9,Mb); {Syntax #3}

PEx explicit operations	Rx = DM(I1,Ma); {Explicit #1}
	Rx = PM(I9,Mb); {Explicit #2}
	Rx = DM(I1,Ma), Rx = PM(I9,Mb); {Explicit #3}

PEy implicit operations	Sx = DM(I1,Ma); {Implicit #1}
	Sx = PM(I9,Mb); {Implicit #2}
	Sx = DM(I1,Ma), Sx = PM(I9,Mb); {Implicit #3}

1.Note that the letters a and b (as in Ma or Mb) indicate numbers for modify registers in DAG1 and DAG2. The letter a indicates a DAG1 register and can be replaced with 0 through 7. The letter b indicates a DAG2 register and can be replaced with 8 through 15.

The PEYEN bit (SISD/SIMD mode select) does not influence broadcast operations. Broadcast loading is particularly useful in SIMD applications where the algorithm needs identical data loaded into each processing element. For more information on SIMD mode (in particular, a list of complementary data registers), see "SIMD (Computational) Operations" on page 2-49.

Alternate (Secondary) DAG Registers

To facilitate fast context switching, the processor includes alternate register sets for all DAG registers. Bits in the MODE1 register control when alternate registers become accessible. While inaccessible, the contents of alternate registers are not affected by processor operations. Note that there is a one cycle latency between writing to MODE1 and being able to access an alternate register set. The alternate register sets for the DAGs are described in this section. For more information on alternate data and results registers, see "Alternate (Secondary) Data Registers" on page 2-39.

Bits in the MODE1 register can activate alternate register sets within the DAGs: the lower half of DAG1 (I, M, L, B0-3), the upper half of DAG1 (I, M, L, B4-7), the lower half of DAG2 (I, M, L, B8-11), and the upper half of DAG2 (I, M, L, B12-15). Figure 4-2 shows the primary and alternate register sets of the DAGs.

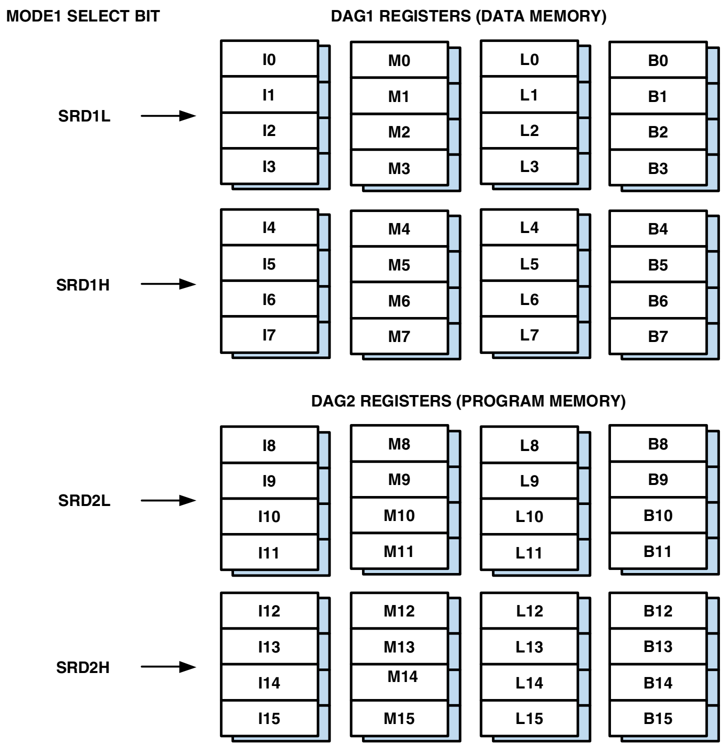


Figure 4-2. Data Address Generator Primary and Alternate Registers

To share data between contexts, a program places the data to be shared in one half of either the current data address generator's registers or the other DAG's registers and activates the alternate register set of the other half. The following example demonstrates how the code handles the one cycle latency from the instruction that sets the bit in MODE1 to when the alternate registers may be accessed. Note that programs can use a NOP instruction or any other instruction not related to the DAG to take care of this latency.

Example 1

```
BIT SET MODE1 SRD1L;   /* Activate alternate dag1 lo regs */
NOP;                   /* Wait for access to alternates */
R0 = DM(i0,m1);
```

Example 2

```
BIT SET MODE1 SRD1L;  /*activate alternate dag1 lo registers */
R13 = R12 + R11;      /* Any unrelated instruction */
R0 = DM(I0,M1);
```

Bit-Reverse Addressing Mode

The BR0 and BR8 bits in the MODE1 register enable the bit-reverse addressing mode where addresses are output in reverse bit order. When BR0 is set (=1), DAG1 bit-reverses 32-bit addresses output from I0. When BR8 is set (=1), DAG2 bit-reverses 32-bit addresses output from I8. The DAGs bit-reverse only the address output from I0 or I8; the contents of these registers are not reversed. Bit-reverse addressing mode effects both pre-modify and post-modify operations. The following example demonstrates how bit-reverse mode effects address output:

```
BIT SET MODE1 BR0;  /* Enables bit-rev. addressing for DAG1 */
I0=0x83000          /* Loads I0 with the bit reverse of the
                       buffer’s base address DM(0xC1000) */

M0 = 0x4000000;     /* Loads M0 with value for post-modify, which
                      is the bit reverse value of the modifier
                      value M0 = 32 */
```

```
R1 = DM(I0,M0);    /* Loads R1 with contents of DM address
                     DM(0xC1000), which is the bit-reverse of
                     0x83000, then post-modifies I0 for the next
                     access with (0x83000 + 0x4000000) =
                     0x4083000, which is the bit-reverse of
                     DM(0xC1020) */
```

In addition to bit-reverse addressing, the processor supports a bit-reverse instruction (BITREV). This instruction bit-reverses the contents of the selected register. For more information on the BITREV instruction, see "Modifying DAG Registers" on page 4-19 or "Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference.

Using DAG Status

The DAGs can provide addressing for a constrained range of addresses, repeatedly cycling through this data (or buffer). A buffer overflow (or wraparound) occurs each time the DAG wraps around the start or end of a buffer's base address. (See "Addressing Circular Buffers" on page 4-13.)

The DAGs can provide buffer overflow information when executing circular buffer addressing for the I7 or I15 registers. When a buffer overflow occurs (a circular buffering operation increments the I register past the end of the buffer or decrements below the start of the buffer), the appropriate DAG updates a buffer overflow flag in a sticky status (STKYx) register. A buffer overflow can also generate a maskable interrupt. Two ways to use buffer overflows from circular buffering are:

- **Interrupts.** Enable interrupts and use an interrupt service routine (ISR) to handle the overflow condition immediately. This method is appropriate if it is important to handle all overflows as they occur; for example in a "ping-pong" or swap I/O buffer pointers routine.

- **STKYx registers.** Use the `BIT TST` instruction to examine overflow flags in the `STKY` register after a series of operations. If an overflow flag is set, the buffer has overflowed—wrapped around—at least once. This method is useful when overflow handling is not time sensitive.

DAG Operations

The processor's DAGs perform several types of operations to generate data addresses. As shown in Figure 4-1, the DAG registers and the `MODE1` and `MODE2` registers contribute to DAG operations. The `STKYx` registers may be affected by the DAG operations and are used to check the status of a DAG operation. The following sections provide details on DAG operations:

- "Addressing With DAGs" on page 4-10
- "Addressing Circular Buffers" on page 4-13
- "Modifying DAG Registers" on page 4-19

An important item to note from Figure 4-1 is that the DAG automatically adjusts the output address per the word size of the address location (short word, normal word, or long word). This address adjustment lets internal memory use the address directly.

SISD/SIMD mode, access word size, and data location (internal) all influence data access operations.

Addressing With DAGs

The DAGs support two types of modified addressing, pre- and post-modify. Modified addressing is used to generate an address that is incremented by a value or a register. In pre-modify addressing, the DAG adds an offset (modifier), which is either an M register or an immediate value, to an I register and outputs the resulting address. Pre-modify addressing does not change or update the I register.

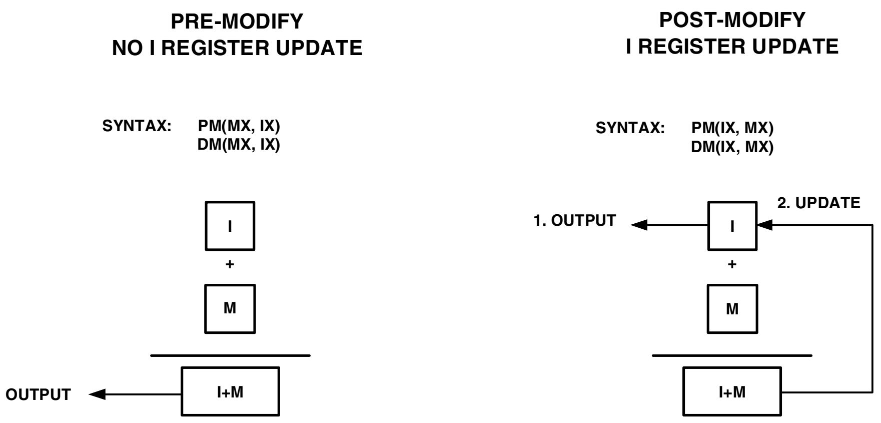


Figure 4-3. Pre-Modify and Post-Modify Operations

In post-modify addressing, the DAG outputs the I register value unchanged, then adds an M register or immediate value, updating the I register value. Figure 4-3 compares pre- and post-modify addressing.

The difference between pre-modify and post-modify instructions in the processor's assembly syntax is the position of the index and modifier in the instruction. If the I register comes before the modifier, the instruction is a post-modify operation. If the modifier comes before the I register, the instruction is a pre-modify without update operation. The following instruction accesses the program memory location indicated by the value in I15 and writes the value I15 + M12 to the I15 register:

```
R6 = PM(I15,M12); /* Post-modify addressing with update */
```

By comparison, the following instruction accesses the program memory location indicated by the value I15 + M12 and does not change the value in I15:

```
R6 = PM(M12,I15); /* Pre-modify addressing without update */
```

Modify (M) registers can work with any index (I) register in the same DAG (DAG1 or DAG2). For a list of I and M registers and their related DAGs, see Figure 4-2 on page 4-7.

Instructions can also use a number (immediate value), instead of an M register, as the modifier. The size of an immediate value that can modify an I register depends on the instruction type. For all single data access operations, modify immediate values can be up to 32 bits wide. Instructions that combine DAG addressing with computations limit the size of the modify immediate value. In these instructions (multifunction computations), the modify immediate values can be up to 6 bits wide. The following example instruction accepts up to 32-bit modifiers:

```
R1 = DM(0x40000000,I1);    /* DM address = I1 + 0x4000 0000 */
```

The following example instruction accepts up to 6-bit modifiers:

```
F6 = F1 + F2,PM(I8,0x0B) = ASTAT; /* PM address = I8,
                                     I8 = I8 + 0x0B */
```

Pre-modify addressing operations must not change the memory space of the address.

Data Addressing Stalls

The instruction sequence stalls for two cycles if a read-after-write hazard is detected on a DAG register. For example, the following sequence automatically generates a two cycle stall.

```
I0 = R0;
DM(I0,M0) <-> R1;
```

If the second instruction is any instruction unrelated to the first instruction, only one cycle stall is inserted.

```
I0 = R0;
R1 = 0X5;          /*Any unrelated instruction */
DM (I0,M0) = R1;  /*Stalls for one cycle */
```

DAG conditional addressing can also generate stalls. These stalls are introduced when the following sequence of instructions is executed. The first is a compute instruction that modifies the `ASTATx`, `ASTATy`, or `FLAGS` registers. The second is a conditional post-modify address generation, and the third is either an address generation operation using the same index register or a read of that index register.

In this sequence, the pipeline is stalled for two cycles.

```
R2 = R3 - R4;           /* Compute setting flags */
IF EQ DM(I1,M1)<-> R1;  /* conditional post-modify addressing */
DM(I1,M2) <-> R2;       /* address generation using the same I
                           register, stalls for two cycles  */
```

When the conditional post-modify instruction is either preceded or followed by instructions other than those involving the address generation using the same `I` register, the last instruction stalls for one cycle. When the conditional post-modify instruction is either preceded or followed by two or more such unrelated instructions, the pipeline does not stall.

Addressing Circular Buffers

The DAGs support addressing circular buffers. This is defined as addressing a range of addresses which contain data that the DAG steps through repeatedly, *wrapping around* to repeat stepping through the range of addresses in a circular pattern. To address a circular buffer, the DAG steps the index pointer (`I` register) through the buffer, post-modifying and updating the index on each access with a positive or negative modify value (`M` register or immediate value). If the index pointer falls outside the buffer, the DAG subtracts or adds the buffer length to the index value, wrapping the index pointer back within the start and end boundaries of the buffer. The DAG's support for circular buffer addressing appears in Figure 4-1 on page 4-3, and an example of circular buffer addressing appears in Figure 4-4 and Figure 4-5.

The starting address that the DAG wraps around is called the buffer's base address (`B` register). There are no restrictions on the value of the base address for a circular buffer.

ⓘ Circular buffering may only use post-modify addressing. The DAG's architecture, as shown in Figure 4-1 on page 4-3, cannot support pre-modify addressing for circular buffering because circular buffering requires that the index be updated on each access.

It is important to note that the DAGs do not detect memory map overflow or underflow. If the address post-modify produces `I + M` > 0xFFFF FFFF or `I - M` < 0, circular buffering may not function correctly. Also, the length of a circular buffer should not let the buffer straddle the top of the memory map. For more information on the processor's memory map, see "ADSP-2136x Memory Maps" on page 5-12.

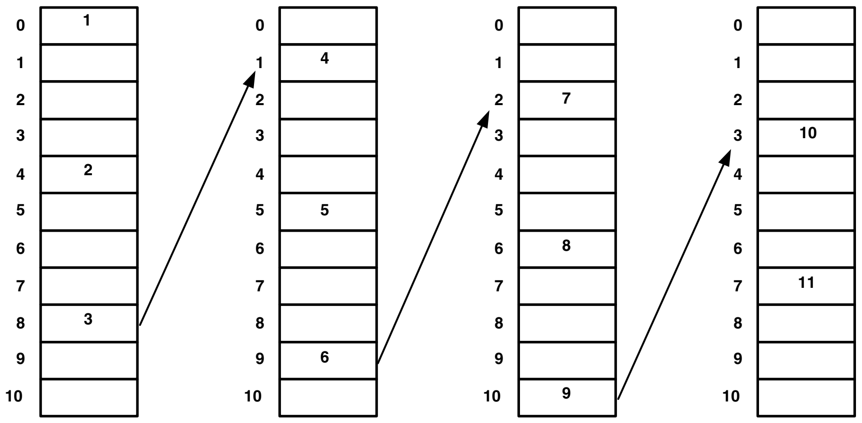
As shown in Figure 4-4, programs use the following steps to set up a circular buffer:

1. Enable circular buffering (`BIT SET MODE1 CBUFEN;`). This operation is only needed once in a program.
2. Load the buffer's base address into the `B` register. This operation automatically loads the corresponding `I` register.
3. Load the buffer's length into the corresponding `L` register. For example, `L0` corresponds to `B0`.

4. Load the modify value (step size) into an `M` register in the corresponding DAG. For example, `M0` through `M7` correspond to `B0`. Alternatively, the program can use an immediate value for the modifier.

```
THE FOLLOWING SYNTAX SETS UP AND ACCESSES A CIRCULAR BUFFER WITH:
LENGTH = 11
BASE ADDRESS = 0X80500
MODIFIER = 4

BIT SET MODE1 CBUFEN;                       /* ENABLES CIRCULAR BUFFER ADDRESSING JUST ONCE IN A PROGRAM */
B0 = 0X80500;                               /* LOADS B0 AND L0 REGISTERS WITH BASE ADDRESS */
L0 = 11;                                    /* LOADS L0 REGISTER WITH LENGTH OF BUFFER */
M1 = 4;                                     /* LOADS M1 WITH MODIFIER OR STEP SIZE */
LCNTR = 11, DO MY_CIR_BUFFER UNTIL LCE;    /* SETS UP A LOOP CONTAINING BUFFER ACCESSES */
R0 = DM(I0,M1);                            /* AN ACCESS WITHIN THE BUFFER USES POST MODIFY ADDRESSING */
        ...                                /* OTHER INSTRUCTIONS IN THE MY_CIR_BUFFER LOOP */
MY_CIR_BUFFER: NOP;                        /* END OF MY_CIR_BUFFER LOOP */
```



THE COLUMNS ABOVE SHOW THE SEQUENCE IN ORDER OF LOCATIONS ACCESSED IN ONE PASS.
NOTE THAT "0" ABOVE IS ADDRESS DM(0X80500). THE SEQUENCE REPEATS ON SUBSEQUENT PASSES.

Figure 4-4. Circular Data Buffers With Positive Modifier

Figure 4-5 shows a circular buffer with a similar syntax as in Figure 4-4, but with a negative modifier.

After circular buffering is set up, the DAGs use the modulus logic in Figure 4-1 on page 4-3 to process circular buffer addressing.

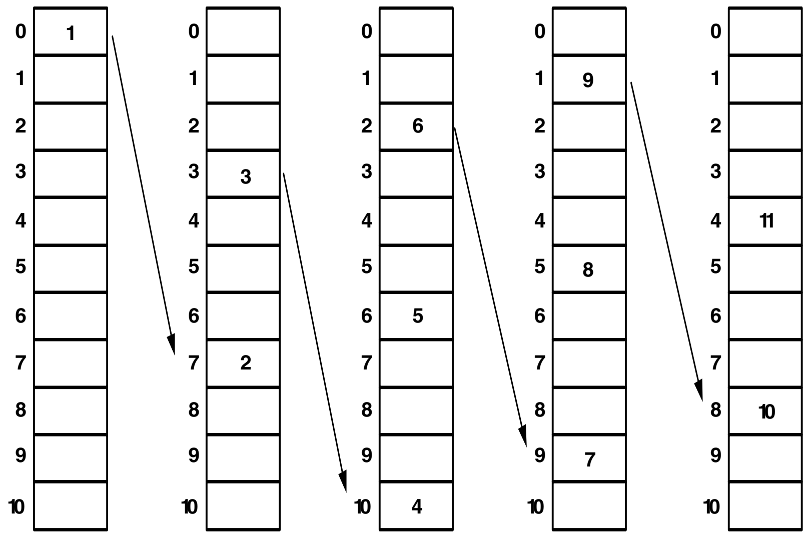


Figure 4-5. Circular Data Buffers With Negative Modifier

On the ADSP-2136x processor, programs enable circular buffering by setting the `CBUFEN` bit in the `MODE1` register. This bit has a corresponding mask bit in the `MMASK` register. Setting the corresponding `MMASK` bit causes the `CBUFEN` bit to be cleared following a push status instruction (`PUSH STS`) or the execution of an external interrupt, timer interrupt, or vectored interrupt. This feature allows programs to disable circular buffering while in an interrupt service routine that does not use circular buffering. By disabling circular buffering, the routine does not need to save and restore the DAG's `B` and `L` registers.

Clearing the `CBUFEN` bit disables circular buffering for all data load and store operations. The DAGs perform normal post-modify load and store accesses, ignoring the `B` and `L` register values. Note that a write to a `B` register modifies the corresponding `I` register, independent of the state of the `CBUFEN` bit. The `MODIFY` instruction executes independent of the state of the CBUFEN bit. The MODIFY instruction always performs circular buffer modify of the index registers if the corresponding B and L registers are configured, independent of the state of the CBUFEN bit.

When circular buffering is enabled, on the first post-modify access to the buffer, the DAG outputs the I register value on the address bus then modifies the address by adding the modify value. If the updated index value is within limits of the buffer, the DAG writes the value to the I register. If the updated value is outside the buffer limits, the DAG subtracts (for positive M) or adds (for negative M) the L register value before writing the updated index value to the I register. In equation form, these post-modify and wraparound operations work as follows.

- If M is positive:

  $I_{new} = I_{old} + M$ if $I_{old} + M <$ Buffer base + length (end of buffer)

  $I_{new} = I_{old} + M - L$ if $I_{old} + M \geq$ buffer base + length

- If M is negative:

  $I_{new} = I_{old} + M$ if $I_{old} + M \geq$ buffer base (start of buffer)

  $I_{new} = I_{old} + M + L$ if $I_{old} + M <$ buffer base (start of buffer)

The DAGs use all four types of DAG registers for addressing circular buffers. These registers operate as follows for circular buffering.

- The index (I) register contains the value that the DAG outputs on the address bus.
- The modify (M) register contains the post-modify value (positive or negative) that the DAG adds to the I register at the end of each memory access. The M register can be any M register in the same DAG as the I register and does not have to have the same number. The modify value can also be an immediate value instead of an M

register. The size of the modify value, whether from an `M` register or immediate, must be less than the length (`L` register) of the circular buffer.

- The length (`L`) register sets the size of the circular buffer and the address range that the DAG circulates the `I` register through. The `L` register must be positive and cannot have a value greater than $2^{31} – 1$. If an `L` register's value is zero, its circular buffer operation is disabled.

- The DAG compares the base (`B`) register, or the `B` register plus the `L` register, to the modified `I` value after each access. When the `B` register is loaded, the corresponding `I` register is simultaneously loaded with the same value. When `I` is loaded, `B` is not changed. Programs can read the `B` and `I` registers independently.

There is one set of registers (`I7` and `I15`) in each DAG that can generate an interrupt on circular buffer overflow (address wraparound). For more information, see "Using DAG Status" on page 4-9.

When a program needs to use `I7` or `I15` without circular buffering and the processor has the circular buffer overflow interrupts unmasked, the program should disable the generation of these interrupts by setting the `B7`/`B15` and `L7`/`L15` registers to values that prevent the interrupts from occurring. If `I7` were accessing the address range 0x1000 – 0x2000, the program could set `B7 =` 0x0000 and `L7 =` 0xFFFF. Because the processor generates the circular buffer interrupt based on the wraparound equations on page 4-17, setting the `L` register to zero does not necessarily achieve the desired results. If the program is using either of the circular buffer overflow interrupts, it should avoid using the corresponding `I` register(s) (`I7` or `I15`) where interrupt branching is not needed.

When working with circular buffers, there are two special situations to be aware of:

1. In the case of circular buffer overflow interrupts, if `CBUFEN` = 1 and register `L7` = 0 (or `L15` = 0), then the `CB7I` (or `CB15I`) interrupt occurs at every change of `I7` (or `I15`), after the index register (`I7` or `I15`) crosses the base register (`B7` or `B15`) value. This behavior is independent of the context of the DAG registers, both primary and alternate.
2. When a long word access, SIMD access, or normal word access with `LW` option crosses the end of the circular buffer, the processor completes the access before responding to the end of buffer condition.

Modifying DAG Registers

The DAGs support two operations that modify an address value in an index register without outputting an address. These two operations, address bit-reversal and address modify, are useful for bit-reverse addressing and maintaining pointers.

The `MODIFY` instruction modifies addresses in any DAG index register (`I0-I15`) without accessing memory.

If the `I` register's corresponding `B` and `L` registers are set up for circular buffering, a `MODIFY` instruction performs the specified buffer wraparound (if needed).

The syntax for `MODIFY` is similar to post-modify addressing (index, then modifier). The `MODIFY` instruction accepts either a 32-bit immediate value or an `M` register as the modifier. The following example adds 4 to `I1` and updates `I1` with the new value:

```
MODIFY(I1,4);
```

The BITREV instruction modifies and bit-reverses addresses in any DAG index register (I0-I15) without accessing memory. This instruction is independent of the bit-reverse mode. The BITREV instruction adds a 32-bit immediate value to a DAG index register, bit-reverses the result, and writes the result back to the same index register. The following example adds 4 to I1, bit-reverses the result, and updates I1 with the new value:

```
BITREV(I1,4);
```

Addressing in SISD and SIMD Modes

Single-instruction, multiple-data (SIMD) mode (PEYEN bit=1) does not change the addressing operations in the DAGs, but it does change the amount of data that moves during each access. The DAGs put the same addresses on the address buses in SIMD and single-instruction single-data (SISD) modes. In SIMD mode, the processor's memory and processing elements get data from the named (explicit) locations in the instruction syntax as well as complementary (implicit) locations. For more information on data moves between registers, see "SIMD (Computational) Operations" on page 2-49.

DAGs, Registers, and Memory

DAG registers are part of the universal register (*Ureg*) set. Programs may load the DAG registers from memory, from another universal register, or with an immediate value. Programs may store DAG registers' contents to memory or to another universal register.

The DAG's registers support the bidirectional register-to-register transfers that are described in "SIMD (Computational) Operations" on page 2-49. When the DAG register is a source of the transfer, the destination can be a register file data register. This transfer results in the contents of the single source register being duplicated in complementary data registers in each processing element.

When the processor is in SIMD mode, if the DAG register is a destination of a transfer from a register file data register source, the processor executes the explicit move only on the condition in PEx becoming true, whereas the implicit move is not performed. This is also true when both the source and the destination is a DAG register.

Programs should use a conditional operation to select either one processing element or neither as the source. Having both processing elements contribute a source value results in the PEx element's write having precedence over the PEy element’s write.

DAG Register-to-Bus Alignment

There are three word alignment types for DAG registers and PM or DM data buses: normal word, extended-precision normal word, and long word.

The DAGs align normal word (32-bit) addressed transfers to the low order bits of the buses. These transfers between memory and 32-bit DAG1 or DAG2 registers use the 64-bit DM and PM data buses. Figure 4-6 illustrates these transfers.

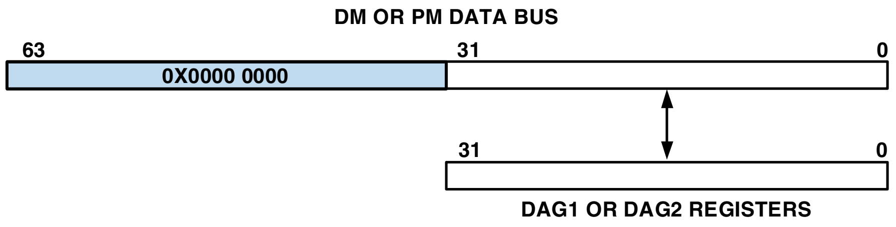


Figure 4-6. Normal Word (32-Bit) DAG Register Memory Transfers

The DAGs align extended-precision normal word (40-bit) addressed transfers or register-to-register transfers to bits 39-8 of the buses. These transfers between a 40-bit data register and 32-bit DAG1 or DAG2 registers use the 64-bit DM and PM data buses. Figure 4-7 illustrates these transfers.

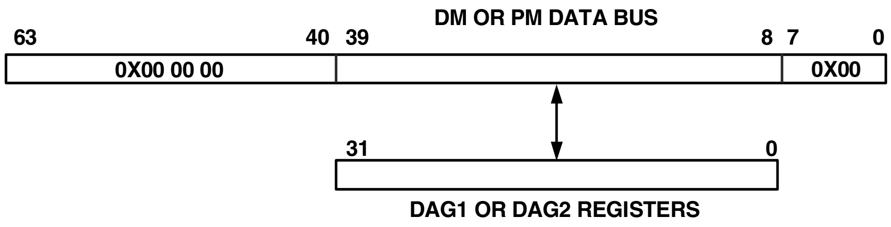


Figure 4-7. DAG Register-to-Data Register Transfers

Long word (64-bit) addressed transfers between memory and 32-bit DAG1 or DAG2 registers target double DAG registers and use the 64-bit DM and PM data buses. Figure 4-8 illustrates how the bus works in these transfers.

If the long word transfer specifies an even numbered DAG register (`I0` or `I2`), then the even numbered register value transfers on the lower half of the 64-bit bus, and the even numbered register + 1 value transfers on the upper half (bits 63-32) of the bus.

If the long word transfer specifies an odd numbered DAG register (`I1` or `B3`), the odd numbered register value transfers on the lower half of the 64-bit bus, and the odd numbered register – 1 value (`I0` or `B2` in this example) transfers on the upper half (bits 63-32) of the bus.

In both the even and odd numbered cases, the explicitly specified DAG register sources or sinks bits 31–0 of the long word addressed memory.

For implicit moves and long word accesses that use the `PX` registers such as `I0 = PX;`, which moves `PX1` to `I0`, only the contents of the `PX1` register are written into `I0`. However, in the code example `PX = I0;` `PX1` and `Px2` are both loaded with `I0`.

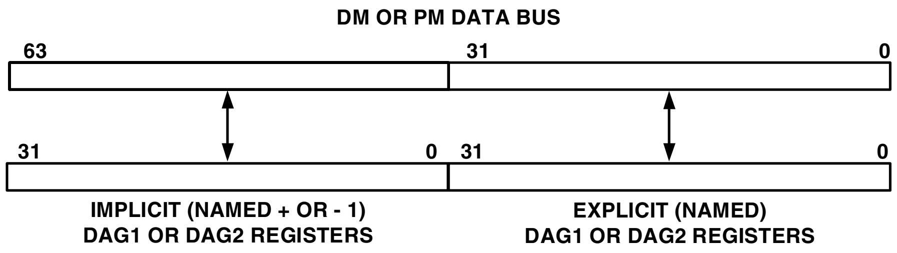


Figure 4-8. Long Word DAG Register-to-Data Register Transfers

DAG Register Transfer Restrictions

The two types of transfer restrictions are hold-off conditions and illegal conditions that the processor does not detect.

Certain sequences of instructions cause incorrect results on the processor and are flagged as errors by the processor assembler software. The following types of instructions can execute on the processor, but cause incorrect results.

- An instruction that stores a DAG register in memory using indirect addressing from the same DAG, with or without an update of the index register. The instruction writes the wrong data to memory or updates the wrong index register.

  Do not try these: `DM(M2,I1) = I0;` or `DM(I1,M2) = I0;`
  These example instructions do not work because `I0` and `I1` are both DAG1 registers.

- An instruction that loads a DAG register from memory using indirect addressing from the same DAG, with an update of the index register. The instruction either loads the DAG register or updates the index register, but not both.

  Do not try this: `L2 = DM(I1,M0);`
  This example instruction does not work because `L2` and `I1` are both DAG1 registers.

DAG Instruction Summary

Table 4-2, Table 4-3, Table 4-4, Table 4-5, Table 4-6, Table 4-7, Table 4-8, and Table 4-9 list the DAG instructions. For more information on assembly language syntax, see "Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference. In these tables, note the meaning of the following symbols:

- `I15-8` indicates a DAG2 index register: `I15`, `I14`, `I13`, `I12`, `I11`, `I10`, `I9`, or `I8`, and `I7-0` indicates a DAG1 index register `I7`, `I6`, `I5`, `I4`, `I3`, `I2`, `I1`, or `I0`.
- `M15-8` indicates a DAG2 modify register: `M15`, `M14`, `M13`, `M12`, `M11`, `M10`, `M9`, or `M8`, and `M7-0` indicates a DAG1 modify register `M7`, `M6`, `M5`, `M4`, `M3`, `M2`, `M1`, or `M0`.
- *`Ureg`* indicates any universal register; for a list of the processor's universal registers, see Table B-1 on page B-2.
- *`Dreg`* indicates any data register; for a list of the processor's data registers, see the Data Register File registers listed in Table B-1 on page B-2.
- *`Data32`* indicates any 32-bit value, and *`Data6`* indicates any 6-bit value.

Table 4-2. Post-Modify Addressing, Modified by M Register and Updating I Register

DM(I7–0,M7–0)=Ureg (LW); {DAG1}
PM(I15–8,M15–8)=Ureg (LW); {DAG2}
Ureg=DM(I7–0,M7–0) (LW); {DAG1}
Ureg=PM(I15–8,M15–8) (LW); {DAG2}
DM(I7–0,M7–0)=Data32; {DAG1}
PM(I15–8,M15–8)=Data32; {DAG2}

Table 4-3. Post-Modify Addressing, Modified by 6-Bit Data and Updating I Register

DM(I7–0,Data6)=Dreg; {DAG1}
PM(I15–8,Data6)=Dreg; {DAG2}
Dreg=DM(I7–0,Data6); {DAG1}
Dreg=PM(I15–8,Data6); {DAG2}

Table 4-4. Pre-Modify Addressing, Modified by M Register (No I Register Update)

DM(M7–0,I7–0)=Ureg (LW); {DAG1}
PM(M15–8,I15–8)=Ureg (LW); {DAG2}
Ureg=DM(M7–0,I7–0) (LW); {DAG1}
Ureg=PM(M15–8,I15–8) (LW); {DAG2}

Table 4-5. Pre-Modify Addressing, Modified by 6-Bit Data (No I Register Update)

DM(Data6,I7–0)=Dreg; {DAG1}
PM(Data6,I15–8)=Dreg; {DAG2}
Dreg=DM(Data6,I7–0); {DAG1}
Dreg=PM(Data6,I15–8); {DAG2}

Table 4-6. Pre-Modify Addressing, Modified by 32-Bit Data (No I Register Update)

Ureg=DM(Data32,I7–0) (LW); {DAG1}
Ureg=PM(Data32,I15–8) (LW); {DAG2}
DM(Data32,I7–0)=Ureg (LW); {DAG1}
PM(Data32,I15–8)=Ureg (LW); {DAG2}

Table 4-7. Update (Modify) I Register, Modified by M Register

Modify(I7–0,M7–0); {DAG1}
Modify(I15–8,M15–8); {DAG2}

Table 4-8. Update (Modify) I Register, Modified by 32-Bit Data

Modify(I7–0,Data32); {DAG1}
Modify(I15–8,Data32); {DAG2}

Table 4-9. Bit-Reverse and Update I Register, Modified By 32-Bit Data

Bitrev(I7–0,Data32); {DAG1}
Bitrev(I15–8,Data32); {DAG2}

5 MEMORY

The ADSP-2136x processors contain four blocks of single-ported internal memory. In most programs this memory is available for single cycle, simultaneous, independent accesses by the core processor and I/O processor. The single-ported memory, in combination with three separate on-chip buses, allows two data transfers from the core and one transfer from the I/O processor in a single cycle, provided the access is from three different blocks of the memory. Using the I/O bus, the I/O processor provides data transfers between internal memory and the processor's communication ports.

As described in the Introduction, the ADSP-2136x processors are comprised of two groups, the ADSP-21362/3/4/5/6 and the ADSP-21367/8/9 processors. The differences in these group's memory is the ADSP-21367/8/9 processors contain an external port which is made up of an asynchronous memory interface, an SDRAM controller and, in the case of the ADSP-21368, a shared memory interface. Also, the processor groups have different amounts of ROM/RAM memory.

This chapter describes the processor's internal and external memory memory and how to use it. For detailed information on the ADSP-21367/8/9 processors external port (and other peripherals), see the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*. For information on connecting and timing accesses to external memory devices that relate to the ADSP-21362/3/4/5/6 processors, see the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors*.

In this chapter both groups are referred to as ADSP-2136x processors. Where differences exist, they are noted explicitly.

The processor memory is organized as four blocks—block 0, block 1, block 2 and block 3, containing up to 3M bits of internal RAM and 6M bits of internal ROM. The memory on the ADSP-2136x processor has the following additional features.

- Each block can be configured for different combinations of code and data storage.
- Each block consists of four columns and each column is 16 bits wide.
- Each block maps to separate regions in memory address space and can be accessed as 16-bit, 32-bit, 48-bit, or 64-bit words.
- Each block also has its own two-deep self clearing shadow write buffers with automatic hit detection and data forwarding logic for read access.
- The processor features a 16-bit floating-point storage format that effectively doubles the amount of data that may be stored on-chip. A single instruction converts the format from 32-bit floating-point to 16-bit floating-point.

While each memory block can store combinations of code and data, accesses are most efficient when one block stores data using the DM bus, for transfers, the second block stores instructions and data using the PM bus and a third and fourth block stores data using the I/O bus. Using the DM and PM buses in this way assures single-cycle execution with two data transfers. In this case, the instruction must be available in the cache.

Internal Memory

The ADSP-2136x processors contain up to 3M bits of internal RAM and up to 6M bits of internal ROM. For information about the maximum number of data or instruction words that can fit into internal memory, see the processor specific data sheet. The current list of titles is located at “Related Documents” on page xxix.

Processor Memory Architecture

Most microprocessors use a single address and a single-data bus for memory accesses. This type of memory architecture is referred to as the Von Neumann architecture. Because processors require greater data throughput than the Von Neumann architecture provides, many processors use memory architectures that have separate data and address buses for program and data storage. These two sets of buses let the processor retrieve a data word and an instruction simultaneously. This type of memory architecture is called Harvard architecture.

SHARC processors go a step further by using a Super Harvard architecture. This four bus architecture has two address buses and two data buses, but provides a single, unified address space for program and data storage. While the data memory (DM) bus only carries data, the program memory (PM) bus handles instructions and data, allowing dual-data accesses.

Processor core and I/O processor accesses to internal memory are completely independent and transparent to one another. Each block of memory can be accessed by the processor core and I/O processor in every cycle provided the access is to different block of the memory.

A memory access conflict can occur when the processor attempts two accesses to the same internal memory block in the same cycle. When this conflict, known as a block conflict occurs, the memory interface logic resolves it according the following rules. The instruction that causes this conflict may take two or three core clock cycles to complete execution.

1. Between DM and PM accesses, conflict is always resolved in favor of DM, with the PM access occurring in the second cycle.
2. Between the core (DM/PM) and I/O accesses, the conflict is resolved in favor of I/O. Note that since the I/O bus runs at half the core clock frequency (`CCLK`), I/O accesses are requested at a maximum rate of once in two core clock cycles. This provides a fair sharing of memory access to the core and I/O buses.

During a single-cycle, dual-data access, the processor core uses the independent PM and DM buses to simultaneously access data from two memory blocks. Though dual-data accesses provide greater data throughput, it is important to note some limitations on how programs may use them. The limitations on single cycle, dual-data accesses are:

- The two pieces of data must come from different memory blocks.

  If the core accesses two words from the same memory block over the same bus in a single instruction, an extra cycle is needed.

- The data access execution may not conflict with an instruction fetch operation. The PM data bus tries to fetch an instruction in every cycle. If a data fetch is also attempted over the PM bus, an extra cycle may be required depending on the cache.

  If the cache contains the conflicting instruction, the data access completes in a single cycle and the sequencer uses the cached instruction. If the conflicting instruction is not in the cache, an extra cycle is needed to complete the data access and cache the conflicting instruction. For more information, see "Instruction Cache" on page 3-8.

For more information on how the buses access memory blocks, see "Internal Memory" on page 5-3.

Buses

As shown in Figure 5-1, the processor has three sets of internal buses connected to its single-ported memory, the program memory (PM), data memory (DM), and I/O processor (IOP) buses. The IOP bus is designed to run only at half the core clock frequency. The three buses share the single port on each of the four memory blocks. Memory accesses from the processor's core (computational units, data address generators, or program sequencer) use the PM or DM buses, while the I/O processor uses the IOP bus for memory accesses. The I/O processor can access external memory devices. For more information about the external memory and I/O capabilities of the processor, see the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21363/4/5/6 Processors* or the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

Internal Address and Data Buses

Figure 5-1 shows that the DM, PM and IOP buses have independent access to internal memory.

Accesses to IOP spaces should not use type 1 (dual access) or LW instructions.

Addresses for the PM and DM buses come from the processor's program sequencer and data address generators (DAGs). The program sequencer generates 24-bit program memory addresses while DAGs supply 32-bit addresses for locations throughout the processor's memory spaces. The DAGs supply addresses for data reads and writes on both the PM and DM address buses, while the program sequencer uses only the PM address bus for sequencing execution.

Each DAG is associated with a particular data bus. DAG1 supplies addresses over the DM bus and DAG2 supplies addresses over the PM bus. For more information on address generation, see "Program Sequencer" on page 3-1 or "Data Address Generators" on page 4-1.

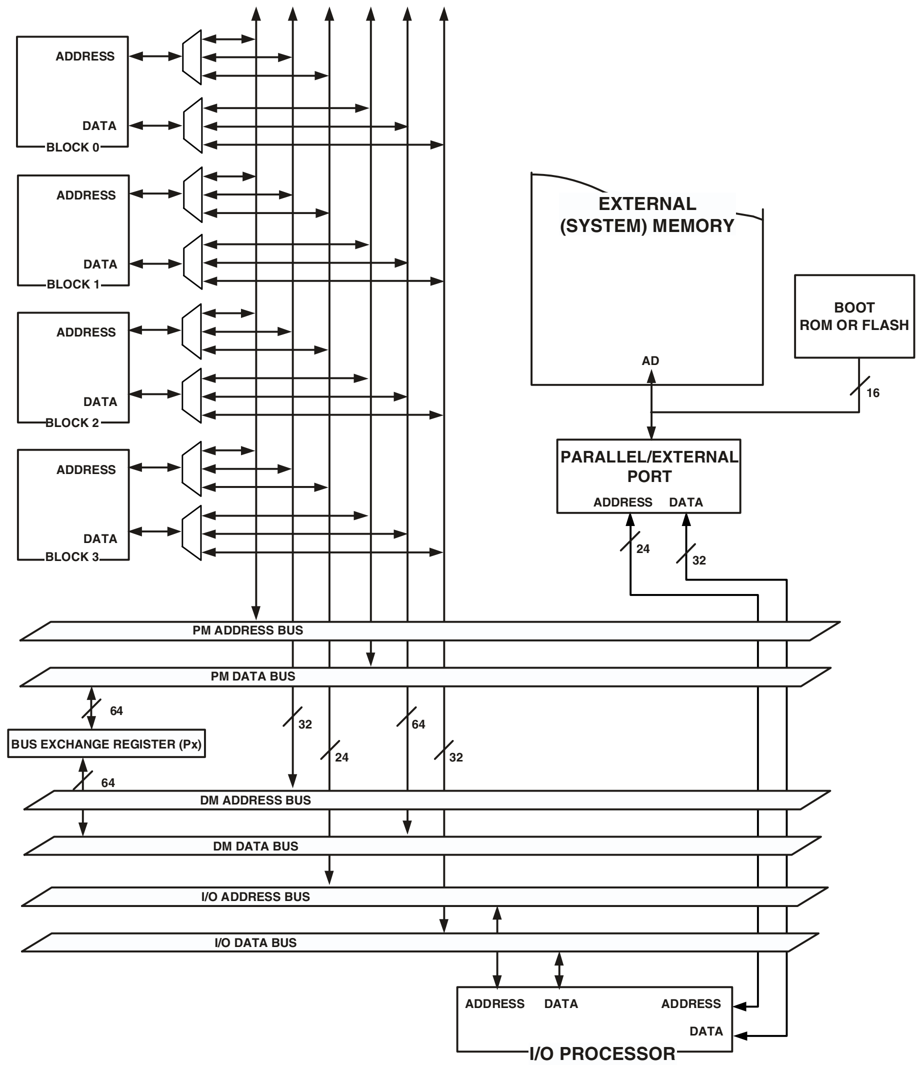


Figure 5-1. Memory and Internal Buses Block Diagram

Because the processor's internal memory is arranged in four 16-bit wide by 64K columns, memory is addressable in widths that are multiples of columns up to 64 bits:

1 column = 16-bit words

2 columns = 32-bit words

3 columns = 48-or 40-bit words

4 columns = 64-bit words

For more information on how the processor works with memory words, see "Memory Organization and Word Size" on page 5-19.

The PM and DM data buses are 64 bits wide. Both data buses can handle long word (64-bit), normal word (32-bit), Extended-precision normal word (40-bit), and short word (16-bit) data, but only the PM data bus carries instruction words (48-bit).

Internal Data Bus Exchange

The data buses allow programs to transfer the contents of any register in the processor to any other register or to any internal memory location in a single cycle. As shown in Figure 5-1, the bus exchange (`PX`) register permits data to flow between the PM and DM data buses. The `PX` register can work as one 64-bit register or as two 32-bit registers (`PX1` and `PX2`). The alignment of `PX1` and `PX2` within `PX` appears in Figure 5-2.

The `PX1`, `PX2`, and the combined `PX` registers are universal registers that are accessible for register-to-register or memory-to-register transfers.

The `PX` register-to-register transfers using data registers are either 40-bit transfers for the combined `PX` or 32-bit transfers for `PX1` or `PX2`. Figure 5-3 shows the bit alignment and gives an example of instructions for register-to-register transfers.

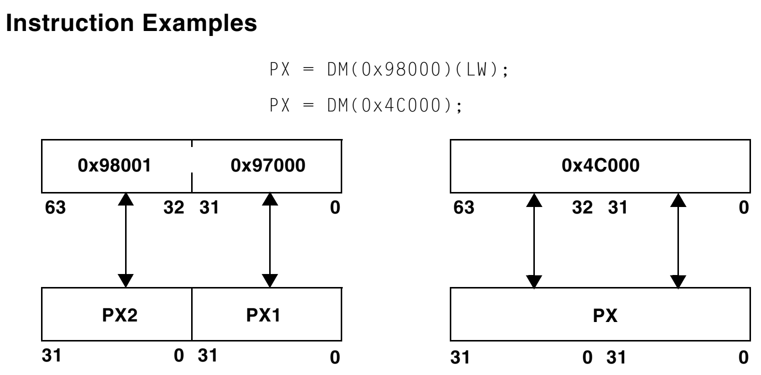


Figure 5-2. PM Bus Exchange (PX, PX1, and PX2) Registers

Figure 5-3 shows that during a transfer between `PX1` or `PX2` and a data register (*`Dreg`*), the bus transfers the upper 32 bits of the register file and zero-fills the eight least significant bits (LSBs).

During a transfer between the combined `PX` register and a register file, the bus transfers the upper 40 bits of `PX` and zero-fills the lower 24 bits.

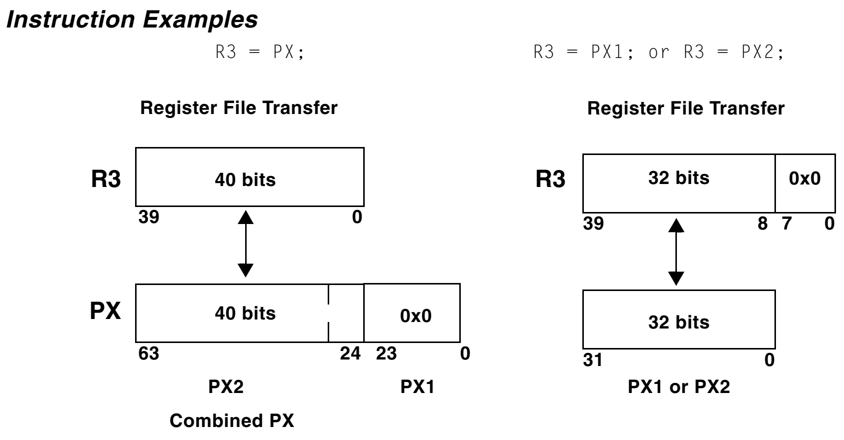


Figure 5-3. PX, PX1, and PX2 Register-to-Register Transfers

The PX register-to-internal memory transfers over the DM or PM data bus are either 48-bit transfers for the combined PX or 32-bit transfers (on bits 31-0 of the bus) for PX1 or PX2. Figure 5-4 shows these transfers.

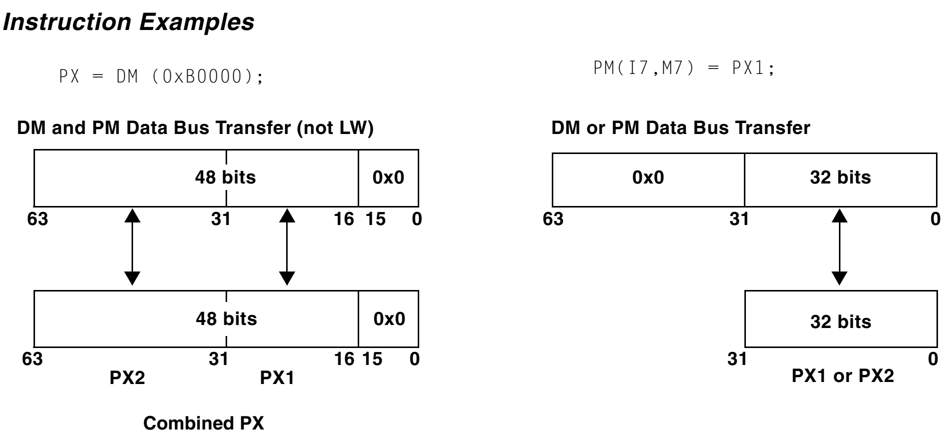


Figure 5-4. PX, PX1, PX2 Register-to-Memory Transfers on DM or PM Data Bus

Figure 5-4 shows that during a transfer between PX1 or PX2 and internal memory, the bus transfers the lower 32 bits of the register.

During a transfer between the combined PX register and internal memory, the bus transfers the upper 48 bits of PX and zero-fills the lower 16 bits.

The status of the memory block's internal memory data width (IMDWx bits in the system control register) setting does not effect this default transfer size for PX to internal memory.

All transfers between the PX register (or any other internal register or memory) and any I/O processor register are 32-bit transfers (least significant 32 bits of PX).

All transfers between the `PX` register and data registers (`R0-R15` or `S0-S15`) are 40-bit transfers. The most significant 40 bits are transferred as shown in Figure 5-3.

Figure 5-5 shows the transfer size between `PX` and internal memory over the PM or DM data bus when using the long word (`LW`) option.

Instruction Example

```
PX = PM (0xB8000)(LW);
```

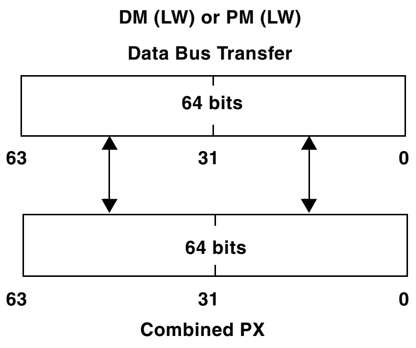


Figure 5-5. PX Register-to-Memory Transfers on PM Data Bus (LW)

The `LW` notation in Figure 5-5 shows an important feature of `PX` register-to-internal memory transfers over the PM or DM data bus for the combined `PX` register. The `PX` register transfers to memory are 48-bit (three column) transfers on bits 63-16 of the PM or DM data bus, unless a long word transfer is used, or the transfer is forced to be 64-bit (four column) with the `LW` (long word) mnemonic. Also note that:

- The `LW` mnemonic affects data accesses that use the `NW` (normal word) addresses irrespective of the settings of the `PEYEN` (processor element Y enable) and `IMDWx` (internal memory data width) bits.

- If a register without a peer such as the PC (program counter) or LCNTR (loop counter) registers, or immediate data is a source for a transfer to a long word memory location, the 32 bit source data is replicated within the long word.

  This is shown in the example below where the long word location 0x4F800 is written with the 64-bit data abbaabba_abbaabba. This is the case for all registers without peers.

  ```
  I0 = 0X4F800;
  M0 = 0X1;
  DM(I0,M0) = 0xabbaabba;
  ```

- Long word accesses with USTATx registers execute as shown below.

  ```
  USTAT1 = DM (LW address);   /* Loads only USTAT1 in SISD
                                 mode */

  DM (LW address) = USTAT1;   /* Stores both USTAT1 and
                                 USTAT2 */
  ```

There is no implicit move when the combined PX register is used in SIMD mode. For example, in SIMD mode, the following moves occur:

```
PX1 = R0;  /* R0 32-bit explicit move to PX1,
              and S0 32-bit implicit move to PX2 */
PX = R0;   /* R0 40-bit explicit move to PX,
              but no implicit move for S0 */
```

However, the following exceptions should be noted:

- Transfers between USTATx and PX registers as in the following example and Figure 5-6. Note that all user status registers behave in this manner.

  ```
  PX = USTAT1;  /* loads PX1 with USTAT1 and PX2 with
                   USTAT2 */
  USTAT1 = PX    /* loads only PX1 to USTAT1 */
  ```

- Transfers between DAG and other system registers and the `PX` register as shown in the following example.

```
I0 = PX         /* Moves PX1 to I0   */
PX = I0         /* Loads both PX1 and PX2 with I0 */
LCNTR = PX      /* Loads LCNTR with PX1 */
PX = PC         /* Loads both PX1 and PX2 with PC */
```

Instruction Example

```
PX = USTAT1;
```

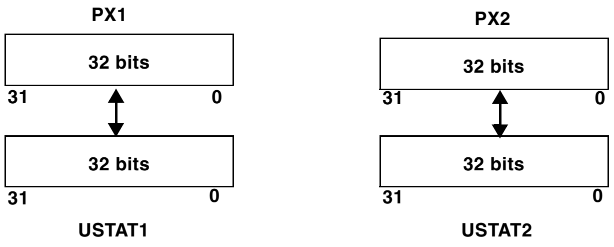


Figure 5-6. Transfers Between USTATx and PX Registers

ADSP-2136x Memory Maps

An example of the ADSP-2136x processor's memory map appears in Table 5-1 and shows three memory spaces: internal memory space, external memory space, and IOP (I/O processor) space. These spaces have these definitions:

- **Internal memory space.** This space ranges from address 0x0004 0000 through 0x0020 000. Internal memory space refers to the processor's on-chip RAM, on-chip ROM, memory-mapped registers and reserved memory space.

- **External memory**. For information on external memory space please refer to the processor specific hardware reference.

- **IOP Space.** This space ranges from address 0x0000 0000 through 0x3FFFF. The I/O processor's memory-mapped registers control the system configuration of the processor and I/O operations. For information about the I/O processor, see the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* or the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors.* These registers occupy consecutive 32-bit locations in this region.

  If a program uses long word addressing (forced with the `LW` mnemonic) to access this region, the access is only to the addressed 32-bit register, rather than the two adjacent I/O processor registers. The register contents are transferred on bits 31–0 of the data bus.

Internal Memory

The ADSP-2136x processors's internal memory space is divided into four blocks—block 0, block 1, block 2 and block 3. RAM and ROM memory varies by processor model and Table 5-1 is just one example. For specific memory organization information, see the processor specific data sheet.

Each block is physically comprised of four 16-bit columns. *Wrapping*, as shown in Figure 5-8 on page 5-21, is a method where memory can efficiently store different combinations of 16-bit, 32-bit, 48-bit or 64-bit wide words. The width of the data word fetched from memory is dependant upon the address range used. The same physical location in memory can be accessed using three different addresses.

For example, the long word address 0x4C000 corresponds to the same locations as normal word address 0x98000 and 0x98001. This also corresponds to the same locations as short word addresses 0x0013 0000, 0x0013 0001, 0x0013 0002 and 0x0013 0003. There are gaps in the memory map when using normal word addressing for 48-bit or 40-bit accesses. These gaps of missing addresses stem from the arrangement of this 3-column data in the memory.

Table 5-1. Example Internal Memory Space (ADSP-21367)

IOP Registers 0x0000 0000–0x0003 FFFF			
Long Word (64 bits)	**Extended-Precision Normal or Instruction Word (48 bits)**	**Normal Word (32 bits)**	**Short Word (16 bits)**
BLOCK 0 ROM 0x0004 0000– 0x0004 BFFF	BLOCK 0 ROM 0x0008 0000– 0x0008 FFFF	BLOCK 0 ROM 0x0008 0000– 0x0009 7FFF	BLOCK 0 ROM 0x0010 0000– 0x0012 FFFF
Reserved 0x0004 F000– 0x0004 FFFF	Reserved 0x0009 4000– 0x0009 FFFF	Reserved 0x0009 E0000– 0x0009 FFFF	Reserved 0x0013 C000– 0x0013 FFFF
BLOCK 0 RAM 0x0004 C000– 0x0004 EFFF	BLOCK 0 RAM 0x0009 0000– 0x0009 3FFF	BLOCK 0 RAM 0x0009 8000– 0x0009 DFFF	BLOCK 0 RAM 0x0013 0000– 0x0013 BFFF
BLOCK 1 ROM 0x0005 0000– 0x0005 BFFF	BLOCK 1 ROM 0x000A 0000– 0x000A FFFF	BLOCK 1 ROM 0x000A 0000– 0x000B 7FFF	BLOCK 1 ROM 0x0014 0000– 0x0016 FFFF
Reserved 0x0005 F000– 0x0005 FFFF	Reserved 0x000B 4000– 0x000B FFFF	Reserved 0x000B E000– 0x000B FFFF	Reserved 0x0017 C000– 0x0017 FFFF
BLOCK 1 RAM 0x0005 C000– 0x0005 EFFF	BLOCK 1 RAM 0x000B 0000– 0x000B 3FFF	BLOCK 1 RAM 0x000B 8000– 0x000B DFFF	BLOCK 1 RAM 0x0017 0000– 0x0017 BFFF
BLOCK 2 RAM 0x0006 0000– 0x0006 0FFF	BLOCK 2 RAM 0x000C 0000– 0x000C 1554	BLOCK 2 RAM 0x000C 0000– 0x000C 1FFF	BLOCK 2 RAM 0x0018 0000– 0x0018 3FFF
Reserved 0x0006 1000– 0x0006 FFFF	Reserved 0x000C 1555– 0x000D FFFF	Reserved 0x000C 2000– 0x000D FFFF	Reserved 0x0018 4000– 0x001B FFFF
BLOCK 3 RAM 0x0007 0000– 0x0007 0FFF	BLOCK 3 RAM 0x000E 0000– 0x000E 1554	BLOCK 3 RAM 0x000E 0000– 0x000E 1FFF	BLOCK 3 RAM 0x001C 0000– 0x001C 3FFF
Reserved 0x0007 1000– 0x0007 FFFF	Reserved 0x000E 1555– 0x000F FFFF	Reserved 0x000E 2000– 0x000F FFFF	Reserved 0x001C 4000– 0x001F FFFF

Accessing a short word memory address accesses one 16-bit word. Consecutive 16-bit short-words are accessed from columns #1, #2, #3, #4, #1 and so on. Accessing a normal word memory address transfers 32 bits (from columns 1 and 2 or 3 and 4). Consecutive 32-bit words are accessed from columns 1 and 2, 3 and 4, 1 and 2 etc. Accessing a long word address transfers 64 bits (from all four columns). For example, the same 16 bits of Block-0 are overwritten in each of the following four write instructions (some, but not all of the short word accesses overwrite more than 16 bits):

Listing 5-1. Overwriting Bits:

```
#include <def2136x.h>
DM(0x0004C000) = PX;   /* long word transfer
                          (64 bits/four columns) */
DM(0x00098000) = R0;   /* normal word transfer
                          (32 bits/two columns) */
DM(0x00130000) = R0;   /* short word transfer
                          (16 bits/1-column) */
USTAT1 = dm(SYSCTL);
bit set USTAT1 IMDW0;   /* set Blk0 access as ext. precision */
dm(SYSCTL) = USTAT1;
DM(0x00090000) = R0;    /* normal word transfer
                           (40 bits/three columns) */
```

Normal word address space is also used by the program sequencer to fetch 48-bit instructions. Note that a 48-bit fetch spans three columns that can lead to a different address range between instruction fetches and data fetches (Figure 5-7 on page 5-20).

Normal word address space can also optionally be used to fetch 40-bit data (from three columns) if the IMDWx (internal memory data width) bit in the SYSCTL register is set. There are four bits in the SYSCTL register, IMDW0-3 that determine whether access to each block is 32 or 40 bits. For more information, see “Accessing Memory” on page 5-31.

The I/O processor's memory-mapped registers control the system configuration of the processor and I/O operations. For information about the I/O processor, see the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21363/4/5/6 Processors* or the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*. These registers occupy consecutive 32-bit locations in this region.

If a program uses long word addressing (forced with the `LW` mnemonic) to access this region, the access is only to the addressed 32-bit register, rather than the two adjacent I/O processor registers. The register contents are transferred on bits 31–0 of the data bus.

Shared Memory

The ADSP-21368 processor supports connecting to common shared external memory with other ADSP-21368 processors to create shared external bus processor systems. This support includes:

- Distributed, on-chip arbitration for the shared external bus
- Fixed and rotating priority bus arbitration
- Bus time-out logic
- Bus lock

For more information, see "External Port" in the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

External Memory

In the ADSP-21367/8/9 processors, the external memory interface supports access to the external memory by direct core accesses and DMA accesses. The external memory address space for non SDRAM addresses is shown in Table 5-2. The external memory address space for SDRAM addresses is shown in Table 5-4. The external memory is divided in to four banks. Any bank can be programmed as asynchronous memory or synchronous memory.

Table 5-2. External Memory Address Space for Non SDRAM Addresses

Bank	Size in words	Address Range
Bank 0	14M	0x0020 0000 – 0x00FF FFFF
Bank 1	16M	0x0400 0000 – 0x04FF FFFF
Bank 2	16M	0x0800 0000 – 0x08FF FFFF
Bank 3	16M	0x0C00 0000 – 0x0CFF FFFF

External memory address space is supported in normal word addressing mode only. Extended-precision, short word and long word addressing modes are not supported. Program execution from external memory is also not supported.

For more information, see “External Port” in the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

External Address Space

The AMI supports 14M words of external memory in bank 1, bank 2, and bank 3 and 12M words of external memory in bank 0. The maximum amount of external data is 64M bytes when the external bus width (set using the `BW` bits 2–1 in the `AMICTL` register) is 32 bits on bank 1, bank 2, or bank 3. When the external bus width is 32 bits or when packing is disabled with other bus widths (`PKDIS` = 1 and `BW` = 16-bit or `PKDIS` = 1 and `BW` = 8-bit) then the external physical memory is the same as the lower 24 bits of the internal physical address, `ADDR23-0` or `ADDR23-0` = internal physical address 23–0.

For an external bus width of 16 bits with packing enabled (PKDIS = 0) the external physical address ADDR23-0 generation is ADDR23-1 = internal physical address 22–0 where ADDR[0] corresponds to the 1st/2nd 16-bit word.

For an external bus width of 8 bits with packing enabled (PKDIS = 0) the external physical address ADDR23-0 generation is ADDR23-2 = internal physical address 21–0 where ADDR1-0 corresponds to the 1st/2nd/3rd/4th 8-bit word. The external physical address map is shown in Table 5-3.

Table 5-3. AMi Address Memory Map

Bus Width	External Memory BANK	External Physical Address (on ADDR23–0)
32-bit (or PKDIS=1)	0	0x20_0000 to 0xFF_FFFF
32-bit (or PKDIS=1)	1, 2 and 3	0x00_0000 to 0xFF_FFFF
16-bit (and PKDIS=0)	0	0x40_0000 to 0xFF_FFFF
16-bit (and PKDIS=0)	1, 2 and 3	0x00_0000 to 0xFF_FFFF
8-bit (and PKDIS=0)	0	0x80_0000 to 0xFF_FFFF
8-bit (and PKDIS=0)	1, 2 and 3	0x00_0000 to 0xFF_FFFF

For more information, see "External Port" in the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

SDRAM Address Mapping

The address that is seen from the processor core and DMA controller is referred as IA31–0 in the following sections. The IA address is divided into three parts to generate the SDRAM row, column and bank addresses.

On the ADSP-21367/8/9 processors, bank 0 starts at address 0x20 0000 in external memory and is followed in order by Banks 1, 2, and 3. When the processor generates an address located within one of the four banks, it asserts the corresponding memory select line, $\overline{\text{MS}}$3-0.

The ranges of memory are shown in Table 5-4 and Table 5-2.

External memory address space is supported in normal word addressing mode only. Extended-precision, short word and long word addressing modes are not supported. Execution from external memory is also not supported.

Table 5-4. External Memory Address Space for SDRAM Addresses

Bank	Size in words	Address Range
Bank 0	62M	0x0020 0000 – 0x03FF FFFF
Bank 1	64M	0x0400 0000 – 0x07FF FFFF
Bank 2	64M	0x0800 0000 – 0x0BFF FFFF
Bank 3	64M	0x0C00 0000 – 0x0FFF FFFF

For more information, see "External Port" in the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

Memory Organization and Word Size

The processor's internal memory is organized as four 16-bit wide by 64K high columns. These columns of memory are addressable as a variety of word sizes:

- 64-bit long word data (four columns)
- 48-bit instruction words or 40-bit extended-precision normal word data (3 columns)
- 32-bit normal word data (2 columns)
- 16-bit short word data (1 column)

Extended-precision normal word (40-bit) data is only accessible if the IMDWx bit is set in the `SYSCTL` register. It is left-justified within a three column location, using bits 47–8 of the location.

Placing 32-Bit and 48-Bit Words

When the processor core or I/O processor addresses memory, the word width of the access determines which columns within the memory are accessed. For instruction word (48 bits) or extended-precision normal word data (40 bits), the word width is 48 bits, and the processor accesses the memory's 16-bit columns in groups of three. Because these sets of three column accesses are packed into a 4 column matrix, there are four rotations of the columns for storing 40- or 48-bit data. The three column word rotations within the four column matrix appear in Figure 5-7.

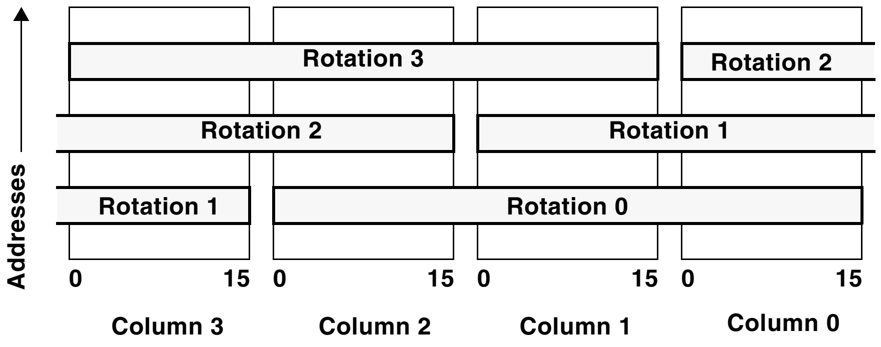


Figure 5-7. 48-Bit Word Rotations

For long word (64 bits), normal word (32 bits), and short word (16 bits) memory accesses, the processor selects from fixed columns in memory. No rotations of words within columns occur for these data types.

Figure 5-8 shows the memory ranges for each data size in the processor's internal memory.

Mixing 32-Bit Words and 48-Bit Words

The processor's memory organization lets programs freely place memory words of all sizes (see "Memory Organization and Word Size" on page 5-19) with few restrictions (see "Restrictions on Mixing 32-Bit Words and 48-Bit Words" on page 5-23). This memory organization also lets programs mix (place in adjacent addresses) words of all sizes. This section discusses how to mix odd (three column) and even (four column) data words in the processor's memory.

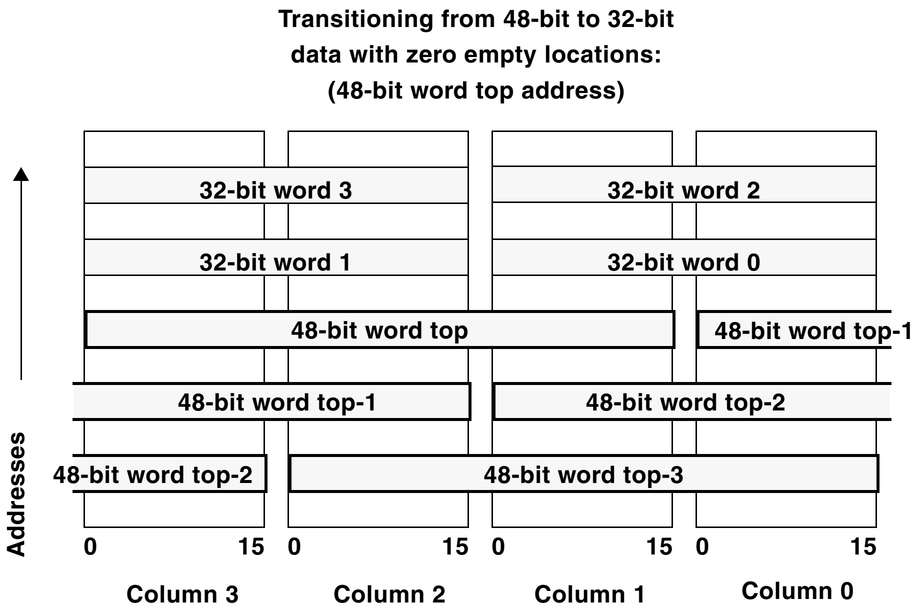


Figure 5-8. Mixed Instructions and Data with No Unused Locations

Transition boundaries between 48-bit (three column) data and any other data size can occur only at any 64-bit address boundary within either internal memory block. Depending on the ending address of the 48-bit words, there are zero, one, or two empty locations at the transition between the 48-bit (three column) words and the 64-bit (four column) words. These empty locations result from the column rotation for storing 48-bit words. The three possible transition arrangements appear in Figure 5-8, Figure 5-9, and Figure 5-10.

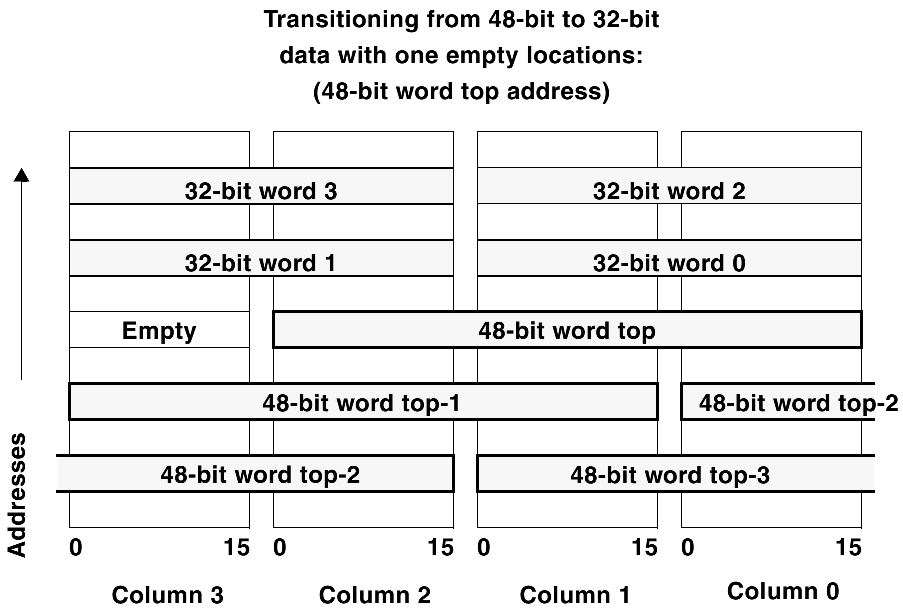


Figure 5-9. Mixed Instructions and Data With One Unused Location

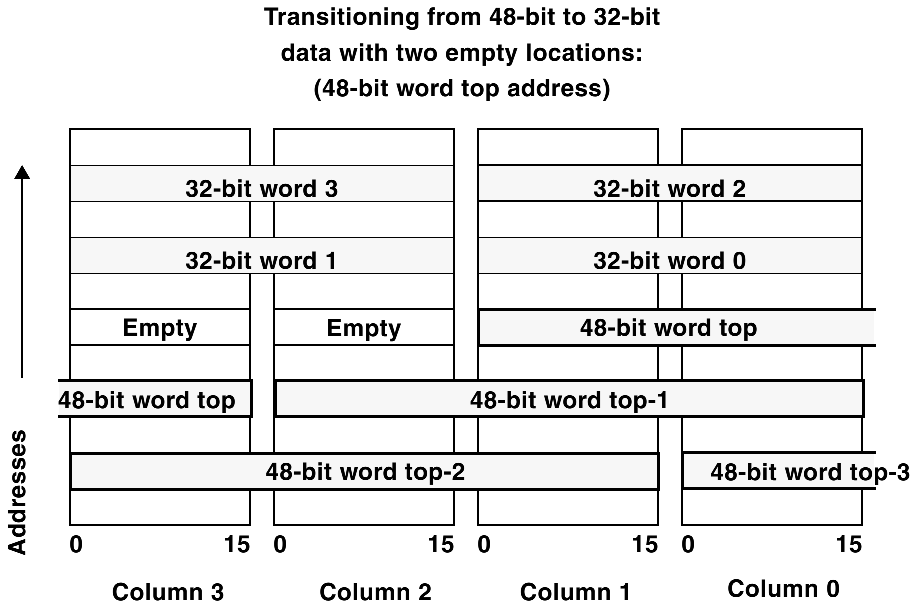


Figure 5-10. Mixed Instructions and Data With Two Unused Locations

Restrictions on Mixing 32-Bit Words and 48-Bit Words

There are some restrictions that stem from the memory column rotations for three column data (48 or 40-bit words) and they relate to the way that three column data can mix with two column data (32-bit words) in memory. These restrictions apply to mixing 48 and 32-bit words, because the processor uses a normal word address to access both of these types of data even though 48-bit data maps onto three columns of memory and 32-bit data maps onto two columns of memory.

When a system has a range of three column (48-bit) words followed by a range of two column (32-bit) words, there is often a gap of empty 16-bit locations between the two address ranges. The size of the address gap varies with the ending address of the range of 48-bit words. Because the addresses within the gap alias to both 48 and 32-bit words, a 48-bit write into the gap corrupts 32-bit locations, and a 32-bit write into the gap corrupts 48-bit locations. The locations within the gap are only accessible with short word (16-bit) accesses.

Calculating the starting address for two column data that minimizes the gap after three column data is useful for programs that are mixing three and two column data. Given the last address of the three column (48-bit) data, the starting address of the 32-bit range that most efficiently uses memory can be determined by the equation:

$m = B + \text{FLOOR}(3/2\ (n - B)) + 1)]$

where:

- **n** is the first unused address after the end of 48-bit words
- **B** is the base normal word address of the internal memory block: **B** = 0x80000 (block 1) else **B = 0xA0000 (Block 2) B** = 0xE0000 (block 3)
- **m** is the first 32-bit normal word address to use after the end of 48-bit words. For

  block 0 = 0x80000 # n # 0x9FFFF

  block 1 = 0xA0000 # n # 0xBFFFF

  block 2 = 0xC0000 # n # 0xDFFFF

  block 3 = 0xE0000 # n # 0xFFFFF

Example: Calculating a Starting Address for 32-Bit Addresses

Given a block of words in the range 0x90000 to 0x92694, the next valid address is 0x82695. The number of 48-bit words (n) is given as follows:

n = 0x92695 - 0x80000 = 0x12695

When 0x12695is converted to decimal representation, the result is 75413.

The base (B) normal word address of the internal memory block is 0x80000. Since `0x88000 < n < 9FFFF2` then the first 32-bit normal word address to use after the end of the 48-bit words is given by:

m = 0x80000 + FLOOR(3/2 (75413)) + 1
m = 0x80000 + 0x1B9E0
m = 0x80000 + 0x1B9E0 = 0x9B9E0

The first valid starting 32-bit address is 0x9B9E0.

48-Bit Word Allocation

Another useful calculation for programs that are mixing two and three column data is to calculate the amount of three column data that minimizes the gap before starting four column data. Given the starting address of the two column (32-bit) data, the number of 48-bit words that most efficiently uses memory can be determined by the equation

n = B + FLOOR (2/3 (m – B)) + 1

where:

- **m** is the first 32-bit normal word address after the end of 32-bit words (1 m values falls in the valid normal word address space)
- **B** is the base normal word address of the internal memory block: **B** = 0x80000 (block 1) else **B = 0xA0000 (Block 2) B** = 0xE0000 (block 3)

- **n** is the address of the first 48-bit word to use after the end of 32-bit words

Using Boot Memory

As shown in Figure 5-1, the processor supports an external boot EPROM or flash. Booting provides the method for automatically loading a program in to the internal memory of the processor after power-up or after a software reset. For information about boot options and the booting process, see the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21363/4/5/6 Processors* or the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

Reading From Boot Memory

When the processor boots from an EPROM, its I/O processor is hard-wired to load 256 instructions (384 32-bit words) automatically from an EPROM (via DMA). Once the initial 256-word DMA is complete, the processor typically needs to maintain access to boot memory, since most programs occupy more then 256 instructions.

Internal Interrupt Vector Table

The default location of the ADSP-2136x processor's interrupt vector table (IVT) depends on the processor's booting mode. When the processor boots from an external source (EPROM, SPI port master or slave booting), the vector table starts at address 0x90000 (normal word). When the processor is in *no boot* mode (runs from internal ROM location 0x80000 without loading), the interrupt vector table starts at address 0x80000.

The internal interrupt vector table (`IIVT`) bit in the `SYSCTL` register overrides the default placement of the vector table. If `IIVT` is set (=1), the interrupt vector table starts at address 0x90000 (internal RAM) regardless of the booting mode.

Internal Memory Data Width

The processor's internal memory blocks use normal word addressing to access either single-precision 32-bit data or extended-precision 40-bit data. Programs select the data width independently for each internal memory block using the internal memory data width (`IMDWx`) bits in the `SYSCTL` register. If a block's `IMDWx` bit is cleared (=0), normal word accesses to the block access 32-bit data. If a block's `IMDWx` bit is set (=1), normal word accesses to the block access 48-bit data. If a program tries to write 40-bit data (for example, a data register-to-memory transfer), the transfer truncates the lower 8 bits from the register, writing only the 32 most significant bits.

If a program tries to read 40-bit data (for example, a memory-to-data register transfer), the transfer zero-fills the lower 8 bits of the register, reading only the 32 most significant bits (MSBs).

The memory bus exchange (`PX`) register is the only exception to these transfer rules—all loads and or stores of the `PX` register are performed as 48-bit accesses unless forced to a 64-bit access with the `LW` mnemonic. If any 40-bit data must be stored in a memory block configured for 32-bit words, the program uses the `PX` register to access the 40-bit data in 48-bit words. Programs should take care not to corrupt any 32-bit data with this type of access. For more information, see "Restrictions on Mixing 32-Bit Words and 48-Bit Words" on page 5-23.

The long word (`LW`) mnemonic only effects normal word address accesses and overrides all other factors (SIMD, `IMDWx`).

Secondary Processor Element (PEy)

When the PEYEN bit in the MODE1 register is set (=1), the processor is in single-instruction, multiple-data (SIMD) mode. In SIMD mode, many data access operations differ from the processor's default single-instruction, single-data (SISD) mode. These differences relate to doubling the amount of data transferred for each data access.

Accesses in SIMD mode transfer both an explicit (named) location and an implicit (unnamed, complementary) location. The explicit transfer is a data transfer between the explicit register and the explicit address, and the implicit transfer is between the implicit register and the implicit address.

For information on complementary (implicit) registers in SIMD mode accesses, see "Secondary Processor Element (PEy)" on page 5-28. For more information on complementary (implicit) memory locations in SIMD mode accesses, see "Accessing Memory" on page 5-31.

Broadcast Register Loads

The processor's BDCST1 and BDCST9 bits in the MODE1 register control broadcast register loading. When broadcast loading is enabled, the processor writes to complementary registers or complementary register pairs in each processing element on writes that are indexed with DAG1 register I1 (if BDCST1 =1) or DAG2 register I9 (if BDCST9 =1). Broadcast load accesses are similar to SIMD mode accesses in that the processor transfers both an explicit (named) location and an implicit (unnamed, complementary) location. However, broadcast loading only influences writes to registers and writes identical data to these registers. Broadcast mode is independent of SIMD mode.

Table 5-5 shows examples of explicit and implicit effects of broadcast register loads to both processing elements. Note that broadcast loading only effects loads of data registers (register file); broadcast loading does not effect register stores or loads to other system registers. Furthermore, broadcast loads only work on register loads; broadcast loading cannot be used for memory writes. For more information on broadcast loading, see "Accessing Memory" on page 5-31.

Table 5-5. Register Load Dual PE Broadcast Operation

Instruction (Explicit, PEx Operation)[1]	(Implicit, PEy operation)

Rx = dm(i1,ma);
Rx = pm(i9,mb);
Rx = dm(i1,ma), Ry = pm(i9,mb);
	Sx = dm(i1,ma);
	Sx = pm(i9,mb);
	Sx = dm(i1,ma), Sy = pm(i9,mb);

1 The post increment in the explicit operation is performed before the implicit instructions are executed.

Illegal I/O Processor Register Access

The processor monitors I/O processor register access when the illegal I/O processor register access (`IIRAE`) bit in the `MODE2` register is set (=1). If access to the IOP registers is detected, an illegal input condition detected (`IICDI`) interrupt occurs. The interrupt is latched in the `IRPTL` register when a core access to an IOP register occurs.

The I/O processor's DMA controller cannot generate the `IICDI` interrupt. For more information, see "Mode Control 2 Register (MODE2)" on page B-11.

Unaligned 64-Bit Memory Access

The processor monitors for unaligned 64-bit memory accesses if the unaligned 64-bit memory accesses (`U64MAE`) bit in the `MODE2` register (bit 21) is set (=1). An unaligned access is an odd numbered address normal word access that is forced to 64 bits with the `LW` mnemonic. When detected, this condition is an input that can cause an illegal input condition detected (IICDI) interrupt if the interrupt is enabled in the IMASK register. For more information, see "Mode Control 2 Register (MODE2)" on page B-11.

The following code example shows the access for even and odd addresses. When accessing an odd address, the sticky bit is set to indicate the unaligned access.

```
bit set mode2 U64MAE;     /* set testbit for aligned or
                             unaligned 64-bit access*/
r0 = 0x11111111;
r1 = 0x22222222;
pm(0x98200) = r0(lw);     /* even address in 32-bit, access
                             is aligned */
pm(0x98201) = r0(lw);     /* odd address in 32-bit, sticky
                              bit is set */
```

Using Memory Access Status

As described in "Illegal I/O Processor Register Access" on page 5-29 and "Unaligned 64-Bit Memory Access" on page 5-29, the processor can provide illegal access information for long word or I/O register accesses. When these conditions occur, the processor updates an illegal condition flag in a sticky status (STKYx) register. Either of these two conditions can also generate a maskable interrupt. Two ways to use illegal access information are:

- **Interrupts.** Enable interrupts and use an interrupt service routine (ISR) to handle the illegal access condition immediately. This method is appropriate if it is important to handle all illegal accesses as they occur.

- **STKYx registers.** Sticky registers hold a value that can be checked for a specific condition at a later time. Use the `Bit Tst` instruction to examine illegal condition flags in the `STKYx` register after an interrupt to determine which illegal access condition occurred.

Accessing Memory

The word width of processor core accesses to internal memory include:

- 48-bit access for instruction words, extended-precision normal word (40-bit) data, and `PX` register
- 64-bit access for long word data, normal word (32-bit) data, or `PX` register data with the `LW` mnemonic
- 32-bit access for normal word (32-bit) data
- 16-bit access for short word data

Long word accesses (both forced and unforced) should not be made to memory-mapped registers.

Accesses to IOP memory spaces should not use type 1 (dual access) or LW instructions.

The processor determines whether a normal word access is 32 or 40 bits from the internal memory block's `IMDWx` setting. For more information, see "Internal Memory Data Width" on page 5-27. While mixed accesses of 48-bit words and 16-, 32-, or 64-bit words at the same address are not allowed, mixed read/writes of 16-, 32-, and 64-bit words to the same address are allowed. For more information, see "Restrictions on Mixing 32-Bit Words and 48-Bit Words" on page 5-23.

The processor's DM and PM buses support 24 combinations of register-to-memory data access options. The following factors influence the data access type:

- Size of words—short word, normal word, extended-precision normal word, or long word
- Number of words—single or dual-data move
- Processor mode—SISD, SIMD, or broadcast load

Access Word Size

The processor's internal memory accommodates the following word sizes:

- 64-bit word data
- 48-bit instruction words
- 40-bit extended-precision normal word data
- 32-bit normal word data
- 16-bit short word data

Long Word (64-Bit) Accesses

A program makes a long word (64-bit) access to internal memory using an access to a long word address. Programs can also make a 64-bit access through normal word addressing with the `LW` mnemonic or through a `PX` register move with the `LW` mnemonic. The address ranges for internal memory accesses appear in Table 5-1 on page 5-14.

When data is accessed using long word addressing, the data is always long word aligned on 64-bit boundaries in internal memory space. When data is accessed using normal word addressing and the `LW` mnemonic, the program should maintain this alignment by using an even normal word address (least significant bit of address = 0). This register selection aligns the normal word address with a 64-bit boundary (long word address) See "Unaligned 64-Bit Memory Access" on page 5-29.

All long word accesses load or store two consecutive 32-bit data values. The register file source or destination of a long word access is a set of two neighboring data registers in a processing element. In a forced long word access (uses the LW mnemonic), the even (normal word address) location moves to or from the explicit register in the neighbor-pair, and the odd (normal word address) location moves to or from the implicit register in the neighbor-pair. For example, the following long word moves could occur:

```
DM(0x98000) = R0 (LW);
/* The data in R0 moves to location DM(0x98000), and the data in
R1 moves to location DM(0x98001) */

R0 = DM(0x98003)(LW);
/* The data at location DM(0x98002) moves to R0, and the data at
location DM(0x98003) moves to R1 */
```

The example shows that R0 and R1 are neighbor registers in the same processing element. Table 5-6 lists the other neighbor register assignments that apply to long word accesses.

In unforced long word accesses (accesses to LW memory space), the processor places the lower 32 bits of the long word in the named (explicit) register and places the upper 32 bits of the long word in the neighbor (implicit) register.

Programs can monitor for unaligned 64-bit accesses by enabling the U64MAE bit. For more information, see "Unaligned 64-Bit Memory Access" on page 5-29.

ⓘ The long word (LW) mnemonic only effects normal word address accesses and overrides all other factors (PEYEN, IMDWx).

Table 5-6. Neighbor Registers for Long Word Accesses

PEx Neighbor Registers	PEy Neighbor Registers
r0 and r1	s0 and s1
r2 and r3	s2 and s3
r4 and r5	s4 and s5
r6 and r7	s6 and s7
r8 and r9	s8 and s9
r10 and r11	s10 and s11
r12 and r13	s12 and s13
r14 and r15	s14 and s15

Instruction Word (48-Bit) and Extended-Precision Normal Word (40-Bit) Accesses

The sequencer uses 48-bit memory accesses for instruction fetches. Programs can make 48-bit accesses with `PX` register moves, which default to 48 bits.

A program makes an extended-precision normal word (40-bit) access to internal memory using an access to a normal word address when that internal memory block's `IMDWx` bit is set (=1) for 40-bit words. The address ranges for internal memory accesses appear in Table 5-1 on page 5-14. For more information on configuring memory for extended-precision normal word accesses, see "Internal Memory Data Width" on page 5-27.

The processor transfers the 40-bit data to internal memory as a 48-bit value, zero-filling the least significant 8 bits on stores and truncating these 8 bits on loads. The register file source or destination of such an access is a single 40-bit data register.

Normal Word (32-Bit) Accesses

A program makes a normal word (32-bit) access to internal memory using an access to a normal word address when that internal memory block's `IMDWx` bit is cleared (=0) for 32-bit words. Programs use normal word addressing to access all processor memory spaces. The address ranges for memory accesses appear in Table 5-1 on page 5-14.

The register file source or destination of a normal word access is a single 40-bit data register. The processor zero-fills the least significant 8 bits on loads and truncates these bits on stores.

Short Word (16-Bit) Accesses

A program makes a short word (16-bit) access to internal memory using an access to a short word address. The address ranges for internal memory accesses appear in Table 5-1 on page 5-14.

The register file source or destination of such an access is a single 40-bit data register. The processor zero-fills the least significant 8 bits on loads and truncates these bits on stores. The 16-bit data occupies bit positions 23–8. Depending on the value of the `SSE` bit in the `MODE1` system register, the processor loads the register's upper 16 bits (bits 39-24) by either:

- Zero-filling these bits if `SSE`=0
- Sign-extending these bits if `SSE`=1

Setting Data Access Modes

The `SYSCTL`, `MODE1` and `MODE2` registers control the operating mode of the processor's memory. The `SYSCTL` register is described in the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21363/4/5/6 Processors* or the *ADSP-2136x SHARC Processor Hardware Reference for the*

ADSP-21367/8/9 Processors. Table B-2 on page B-5 lists the bits in the MODE1 register, and Table B-3 on page B-11 lists the bits in the MODE2 register.

SYSCTL Register Control Bits

The following bits in the SYSCTL register control memory access modes:

- **Internal interrupt vector table.** SYSCTL Bit 2 (IIVT) forces placement of the interrupt vector table at address 0x90000 regardless of booting mode (if 1) or allows placement of the interrupt vector table as selected by the booting mode (if 0).
- **Internal memory block data width.** SYSCTL Bits 12-9 (IMDWx) select the normal word data access size for internal memory block 0, block1, block 2 and block 3. A block's normal word access size is fixed as 32 bits (two column, IMDWx=0) or 48 bits (three column, IMDWx = 1).

Mode 1 Register Control Bits

The following bits in the MODE1 register control memory access modes:

- **Secondary processor element (PEy).** MODE1 Bit 21 (PEYEN) enables computations in PEy (SIMD mode), (if 1) or disables PEy (SISD mode), (if 0).
- **Broadcast register loads.** MODE1 Bit 22 (BDCST9) and Bit 23 (BDCST1) enable broadcast register loads for memory transfers indexed with I1 (if BDCST1 = 1) or indexed with I9 (if BDCST9 =1).

Mode 2 Register Control Bits

The following bits in the `MODE2` register control memory access modes:

- **Illegal IOP register access enable.** `MODE2` Bit 20 (`IIRAE`) enables detection of IOP register access (if 1) or disables detection (if 0).
- **Unaligned 64-bit memory access enable.** `MODE2` Bit 21 (`U64MAE`) enables detection of uneven address memory access (if 1) or disables detection (if 0).

SISD, SIMD, and Broadcast Load Modes

These modes influence memory accesses. For a comparison of their effects, see the examples in "Data Access Options" on page 5-38. and "Secondary Processing Element (PEy)" on page 2-45.

Broadcast load mode is a hybrid between SISD and SIMD modes that transfers dual-data under special conditions. For examples of broadcast transfers, see "Data Access Options" on page 5-38. For more information on broadcast load mode, see "Broadcast Register Loads" on page 5-28.

Single- and Dual-Data Accesses

The number of transfers that occur in a cycle influences the data access operation. As described in "Processor Memory Architecture" on page 5-3, the processor supports single cycle, dual-data accesses to and from internal memory for register-to-memory and memory-to-register transfers. Dual-data accesses occur over the PM and DM bus and act independent of SIMD/SISD. Though only available for transfers between memory and data registers, dual-data transfers are extremely useful because they double the data throughput over single-data transfers.

Instruction Examples

```
R8 = DM (I4,M3), PM (I12,M13) = R0; /* Dual access */
R0 = DM (I5,M5);                    / * Single access */
```

For examples of data flow paths for single and dual-data transfers, see the following section, "Data Access Options".

Data Access Options

The following list shows the processor's possible memory transfer modes and provides a cross-reference to examples of each memory access option that stems from the processor's data access options.

These modes include the transfer options that stem from the following data access options:

- The mode of the processor: SISD, SIMD, or Broadcast Load
- The size of access words: long, extended-precision normal word, normal word, or short word
- The number of transferred words

To take advantage of the processor's data accesses to three and four column locations, programs must adjust the interleaving of data into memory locations to accommodate the memory access mode. The following guidelines provide overviews of how programs should interleave data in memory locations. For more information and examples, see "Instruction Set" in Chapter 8, Instruction Set, and "Computations Reference" in Chapter 9, Computations Reference.

- Programs can use odd or even modify values (1, 2, 3, …) to step through a buffer in single- or dual-data, SISD or broadcast load mode regardless of the data word size (long word, extended-precision normal word, normal word, or short word).

- Programs should use a multiple of 4 modify values (4, 8, 12, ...) to step through a buffer of short word data in single- or dual-data, SIMD mode. Programs must step through a buffer twice, once for addressing even short word addresses and once for addressing odd short word addresses.
- Programs should use a multiple of 2 modify values (2, 4, 6, ...) to step through a buffer of normal word data in single- or dual-data SIMD mode.
- Programs can use odd or even modify values (1, 2, 3, ...) to step through a buffer of long word or extended-precision normal word data in single- or dual-data SIMD modes.

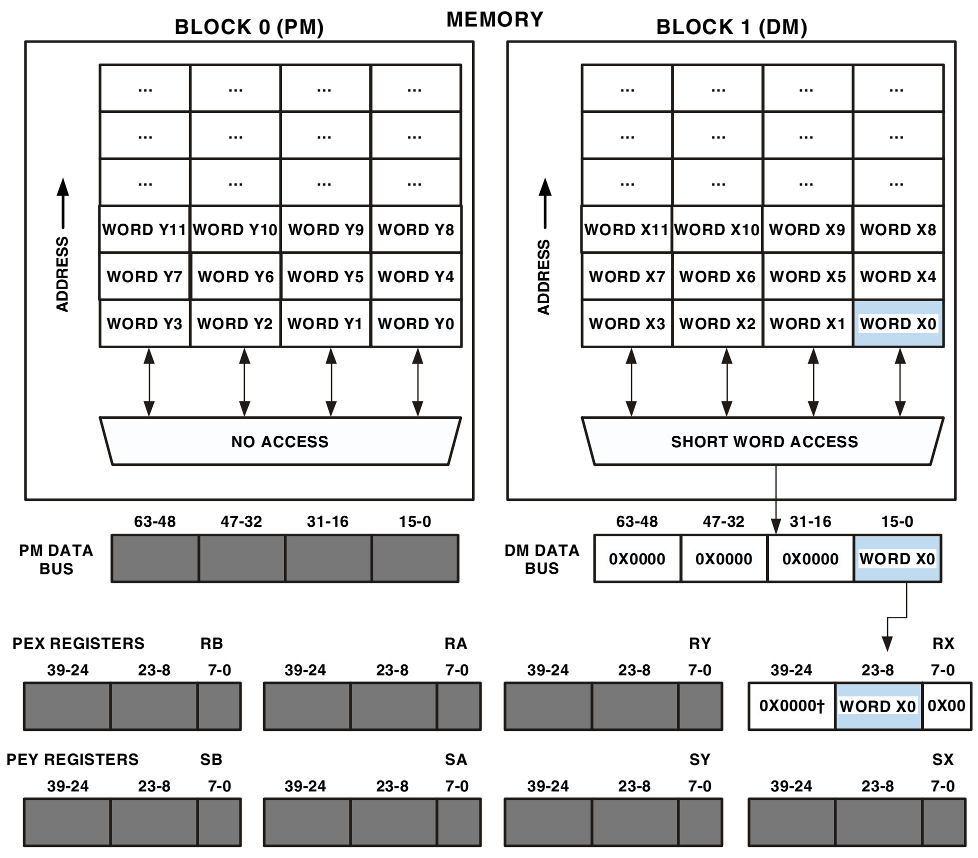
Short Word Addressing of Single-Data in SISD Mode

Figure 5-11 shows the SISD single-data, short word addressed access mode. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The 16-bit value for the short word access is transferred using the least significant short word lane of the PM or DM data bus. The processor drives the other short word lanes of the data buses with zeros.

In SISD mode, the instruction accesses the PEx registers to transfer data from memory. This instruction accesses WORD X0, whose short word address has "00" for its least significant two bits of address. Other locations within this row have addresses with least significant two bits of "01", "10", or "11" and select WORD X1, WORD X2, or WORD X3 from memory respectively. The syntax targets register RX in PEx. The example in Figure 5-11 targets a PEy register using the syntax SX.

The cross (†) in the PEx registers in Figure 5-11 indicates that the processor zero-fills or sign-extends the most significant 16 bits of the data register while loading the short word value into a 40-bit data register.

Zero-filling or sign-extending depends on the state of the SSE bit in the MODE1 system register. For short word transfers, the least significant 8 bits of the data register are always zero.



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(SHORT WORD X0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, SHORT WORD, SINGLE-DATA TRANSFERS ARE:

|UREG = PM(SHORT WORD ADDRESS);|
|UREG = DM(SHORT WORD ADDRESS);|
|PM(SHORT WORD ADDRESS) = UREG;|
|DM(SHORT WORD ADDRESS) = UREG;|

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-11. Short Word Addressing of Single-Data in SISD Mode

Short Word Addressing of Single-Data in SIMD Mode

Figure 5-12 shows the SIMD, single-data, short word addressed access mode. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The explicitly addressed (named in the instruction) 16-bit value is transferred using the least significant short word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) short word value is transferred using the 47-32 bit short word lane of the PM or DM data bus. The processor drives the other short word lanes of the PM or DM data buses with zeros (31-16 bit lane and 63-48 bit lane).

The instruction explicitly accesses the register RX and implicitly accesses that register's complementary register, SX. This instruction uses a PEx register with an RX mnemonic. If the syntax named the PEy register SX as the explicit target, the processor uses that register's complement RX as the implicit target. For more information on complementary registers, see "Secondary Processing Element (PEy)" on page 2-45.

The cross (†) in the PEx and PEy registers in Figure 5-12 indicates that the processor zero-fills or sign-extends the most significant 16 bits of the data register while loading the short word value into a 40-bit data register. Zero-filling or sign-extending depends on the state of the SSE bit in the MODE1 system register. For short word accesses, the least significant 8 bits of the data register are always zero.

Figure 5-12 shows the data path for one transfer. The processor accesses short words sequentially in memory. Table 5-7 shows the pattern of SIMD mode short word accesses. For more information on arranging data in memory to take advantage of this access pattern, see Figure 5-33 on page 5-79.

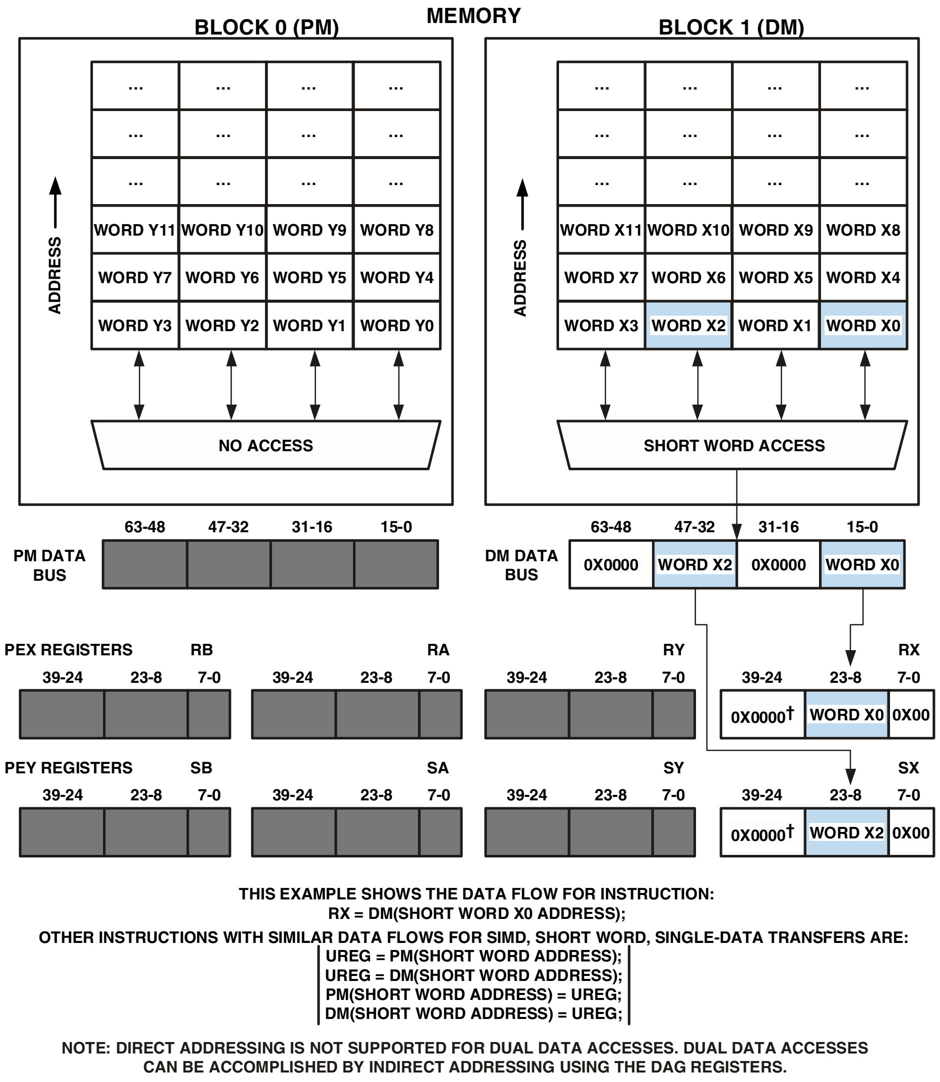

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y11 WORD Y10 WORD Y9 WORD Y8
WORD Y7 WORD Y6 WORD Y5 WORD Y4
WORD Y3 WORD Y2 WORD Y1 WORD Y0
WORD X11 WORD X10 WORD X9 WORD X8
WORD X7 WORD X6 WORD X5 WORD X4
WORD X3 WORD X2 WORD X1 WORD X0
NO ACCESS
SHORT WORD ACCESS
PM DATA BUS
DM DATA BUS
63-48 47-32 31-16 15-0
0X0000 WORD X2 0X0000 WORD X0
PEX REGISTERS
PEY REGISTERS
RB RA RY RX
SB SA SY SX
39-24 23-8 7-0
0X0000† WORD X0 0X00
0X0000† WORD X2 0X00
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(SHORT WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, SHORT WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(SHORT WORD ADDRESS);
UREG = DM(SHORT WORD ADDRESS);
PM(SHORT WORD ADDRESS) = UREG;
DM(SHORT WORD ADDRESS) = UREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

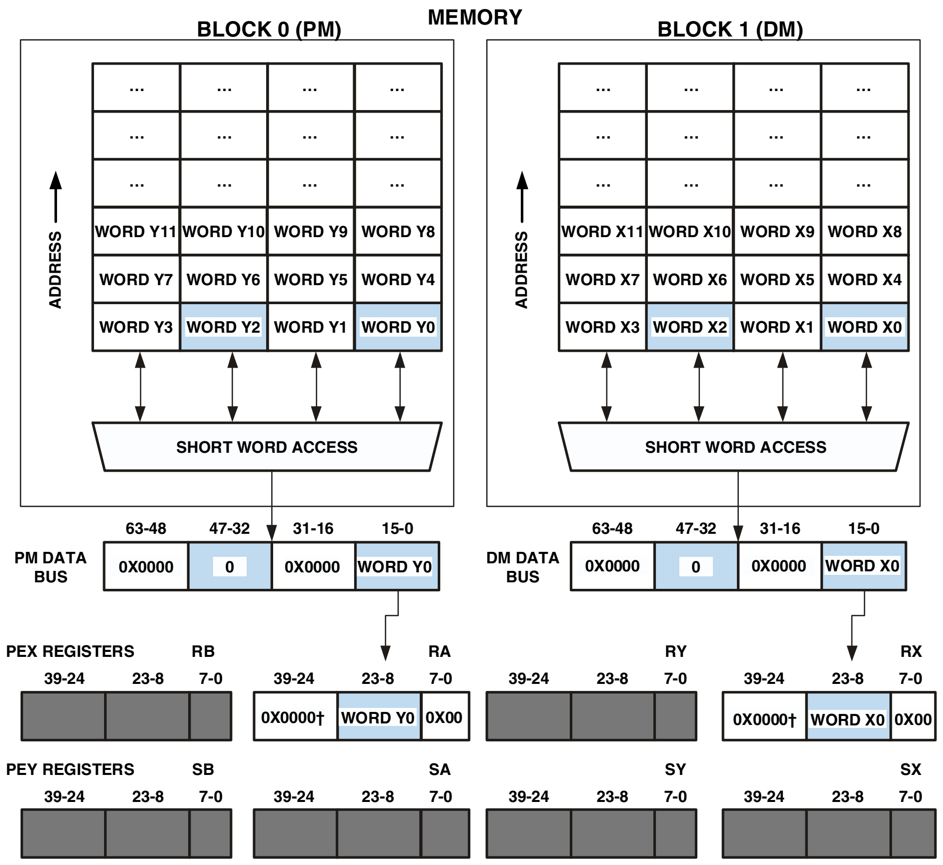
Figure 5-12. Short Word Addressing of Single-Data in SIMD Mode

Short Word Addressing of Dual-Data in SISD Mode

Figure 5-13 shows the SISD, dual-data, short word addressed access mode. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The 16-bit values for short word accesses are transferred using the least significant short word lanes of the PM and DM data buses. The processor drives the other short word lanes of the data buses with zeros. Note that the accesses on both buses do not have to be the same word width. SISD mode dual-data accesses can handle any combination of short word, normal word, extended-precision normal word, or long word accesses. For more information, see "Mixed-Word Width Addressing of Dual-Data in SISD Mode" on page 5-68.

In SISD mode, the instruction explicitly accesses PEx registers. This instruction accesses WORD X0 in block 1 and WORD Y0 in block 0. Each of these words has a short word address with "00" for its least significant two bits of address. Other accesses within these four column locations have addresses with their least significant two bits as "01", "10", or "11" and select WORD X1/Y1, WORD X2/Y2, or WORD X3/Y3 from memory respectively. The syntax explicitly accesses registers RX and RY in PEx. The example targets PEy registers when using the syntax SX or SY.

The cross (†) in the PEx registers in Figure 5-13 indicates that the processor zero-fills or sign-extends the most significant 16 bits of the data register while loading a short word value into a 40-bit data register. Zero-filling or sign-extending depends on the state of the SSE bit in the MODE1 system register. For short word accesses, the least significant 8 bits of the data register are always zero.



THE ABOVE EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(SHORT WORD X0 ADDRESS), RA = PM(SHORT WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, SHORT WORD, DUAL-DATA TRANSFERS ARE:
|DREG = PM(SHORT WORD ADDRESS), | DREG = DM(SHORT WORD ADDRESS);|
|PM(SHORT WORD ADDRESS) = DREG, | DM(SHORT WORD ADDRESS) = DREG;|

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-13. Short Word Addressing of Dual-Data in SISD Mode

Short Word Addressing of Dual-Data in SIMD Mode

Figure 5-14 shows the SIMD, dual-data, short word addressed access. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The explicitly addressed 16-bit values are transferred using the least significant short word lanes of the PM and DM data bus. The implicitly addressed short word values are transferred using the 47-32 bit short word lanes of the PM and DM data buses. The processor drives the other short word lanes of the PM and DM data buses with zeros.

The accesses on both buses do not have to be the same word width. SIMD mode dual-data accesses can handle combinations of short word and normal word or extended-precision normal word and long word accesses. For more information, see "Mixed-Word Width Addressing of Dual-Data in SIMD Mode" on page 5-70.

The instruction explicitly accesses registers RX and RA, and implicitly accesses the complementary registers, SX and SA. This instruction uses PEx registers with the RX and RA mnemonics. If the syntax named PEy registers SX and SA as the explicit targets, the processor uses those registers' complements, RX and RA, as the implicit targets. For more information on complementary registers, see "Secondary Processing Element (PEy)" on page 2-45.

The cross (†) in the PEx and PEy registers in Figure 5-14 indicates that the processor zero-fills or sign-extends the most significant 16 bits of the data registers while loading the short word values into the 40-bit data registers. For short word accesses, zero-filling or sign-extending depends on the state of the SSE bit in the MODE1 system register. For the short word accesses, the least significant 8 bits of the data register are always zero.

The second word from Block 1 is shown as x2 on the data bus and in the Sx register. It is shown as Y2 and Y1 respectively. The Sx and SA registers are transparent and look similar to Rx and RA. All bits should be shown as in Rx and RA. For more information on arranging data in memory to take advantage of short word addressing of dual-data in SIMD mode, see Figure 5-34 on page 5-80.

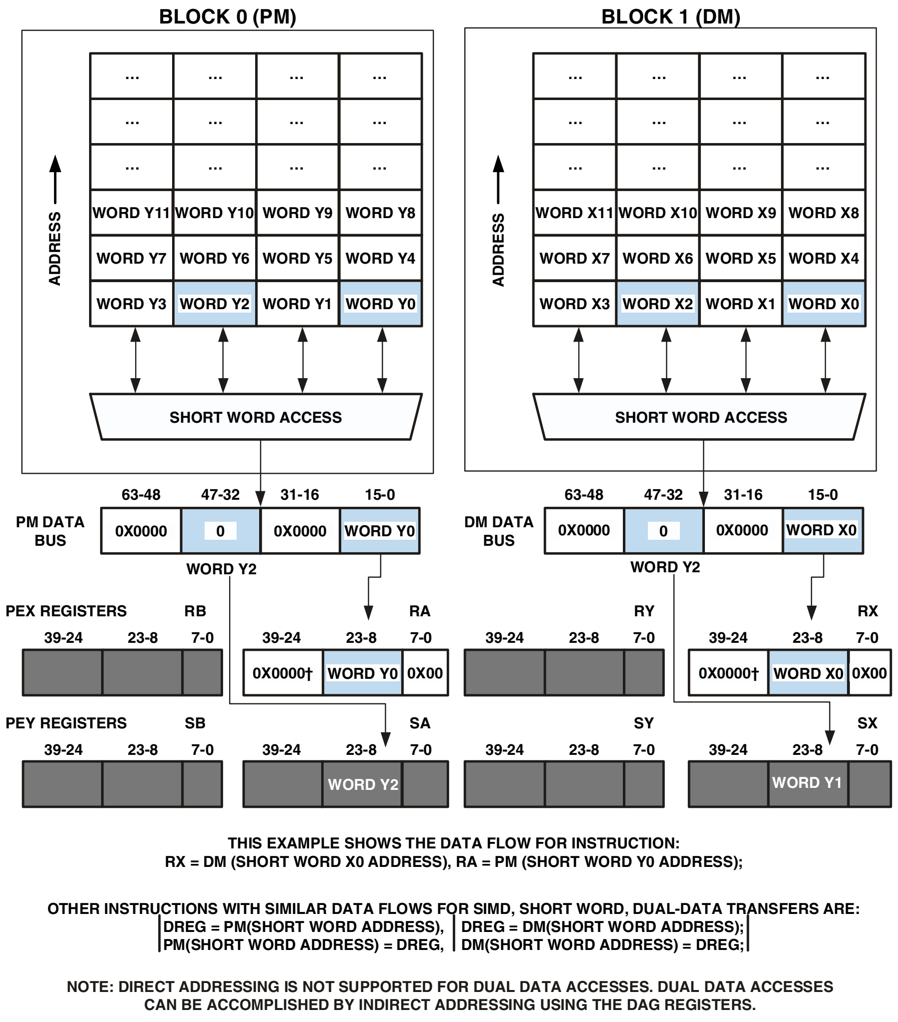

BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y11
WORD Y10
WORD Y9
WORD Y8
WORD Y7
WORD Y6
WORD Y5
WORD Y4
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X11
WORD X10
WORD X9
WORD X8
WORD X7
WORD X6
WORD X5
WORD X4
WORD X3
WORD X2
WORD X1
WORD X0
SHORT WORD ACCESS
PM DATA BUS
DM DATA BUS
63-48
47-32
31-16
15-0
0X0000
0
WORD Y2
PEX REGISTERS
PEY REGISTERS
RB
RA
SB
SA
RY
RX
SY
SX
39-24
23-8
7-0
0X0000†
0X00
WORD Y1
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM (SHORT WORD X0 ADDRESS), RA = PM (SHORT WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, SHORT WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(SHORT WORD ADDRESS), DREG = DM(SHORT WORD ADDRESS);
PM(SHORT WORD ADDRESS) = DREG, DM(SHORT WORD ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-14. Short Word Addressing of Dual-Data in SIMD Mode

32-Bit Normal Word Addressing of Single-Data in SISD Mode

Figure 5-15 shows the SISD, single-data, 32-bit normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The 32-bit value for the normal word access completes a transfer using the least significant normal word lane of the PM or DM data bus. The processor drives the other normal word lanes of the data buses with zeros.

In SISD mode, the instruction accesses a `PEx` register. This instruction accesses `WORD X0` whose normal word address has “0” for its least significant address bit. The other access within this four column location has an address with a least significant bit of “1” and selects `WORD X1` from memory. The syntax targets register `RX` in `PEx`. The example targets a `PEy` register when using the syntax `SX`.

For normal word accesses, the processor zero-fills the least significant 8 bits of the data register on loads and truncates these bits on stores to memory.

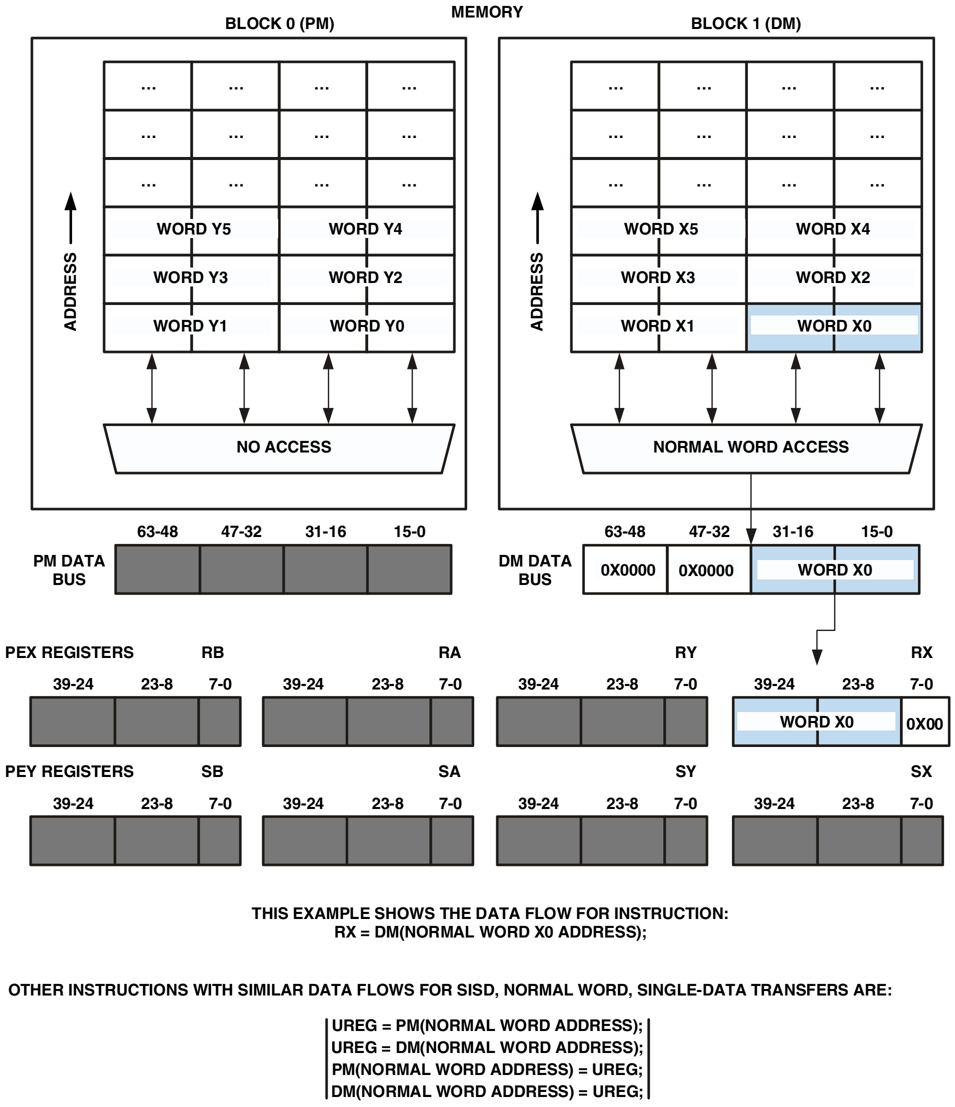

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y5
WORD Y4
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X5
WORD X4
WORD X3
WORD X2
WORD X1
WORD X0
NO ACCESS
NORMAL WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
0X0000
0X0000
WORD X0
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
WORD X0
0X00
PEY REGISTERS
SB
SA
SY
SX
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, NORMAL WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(NORMAL WORD ADDRESS);
UREG = DM(NORMAL WORD ADDRESS);
PM(NORMAL WORD ADDRESS) = UREG;
DM(NORMAL WORD ADDRESS) = UREG;

Figure 5-15. Normal Word Addressing of Single-Data in SISD Mode

32-Bit Normal Word Addressing of Single-Data in SIMD Mode

Figure 5-16 shows the SIMD, single-data, normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The explicitly addressed (named in the instruction) 32-bit value completes a transfer using the least significant normal word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) normal word value completes a transfer using the most significant normal word lane of the PM or DM data bus.

In Figure 5-16, the explicit access targets the named register `RX`, and the implicit access targets that register's complementary register, `SX`. This instruction uses a `PEx` register with an `RX` mnemonic. If the syntax named the `PEy` register `SX` as the explicit target, the processor would use that register's complement, `RX`, as the implicit target. For more information on complementary registers, see "Secondary Processing Element (PEy)" on page 2-45.

For normal word accesses, the processor zero-fills the least significant 8 bits of the data register on loads and truncates these bits on stores to memory.

Figure 5-16 shows the data path for one transfer. The processor accesses normal words sequentially in memory (see Table 5-7). For more information on arranging data in memory to take advantage of this access pattern, see Figure 5-34 on page 5-80.

Table 5-7. Normal Word Addressing in SIMD Mode

Explicit Normal Word Accessed	Implicit Normal Word Accessed
Word X0 (address LSB = 0)	Word X1 (address LSB = 1)
Word X1 (address LSB = 1)	Word X2 (address LSB = 0)

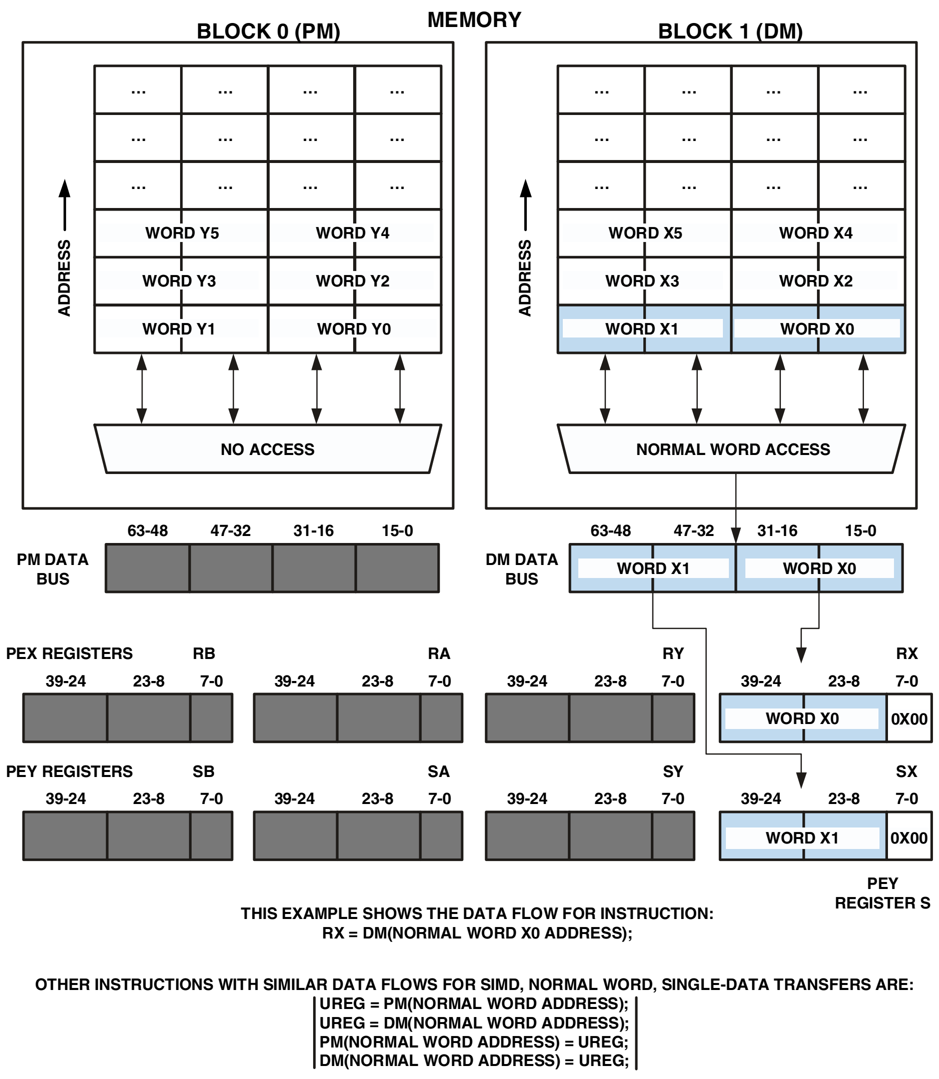

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
...
WORD Y5
WORD Y4
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X5
WORD X4
WORD X3
WORD X2
WORD X1
WORD X0
NO ACCESS
NORMAL WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
0X00
PEY REGISTERS
SB
SA
SY
SX
PEY REGISTER S
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, NORMAL WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(NORMAL WORD ADDRESS);
UREG = DM(NORMAL WORD ADDRESS);
PM(NORMAL WORD ADDRESS) = UREG;
DM(NORMAL WORD ADDRESS) = UREG;

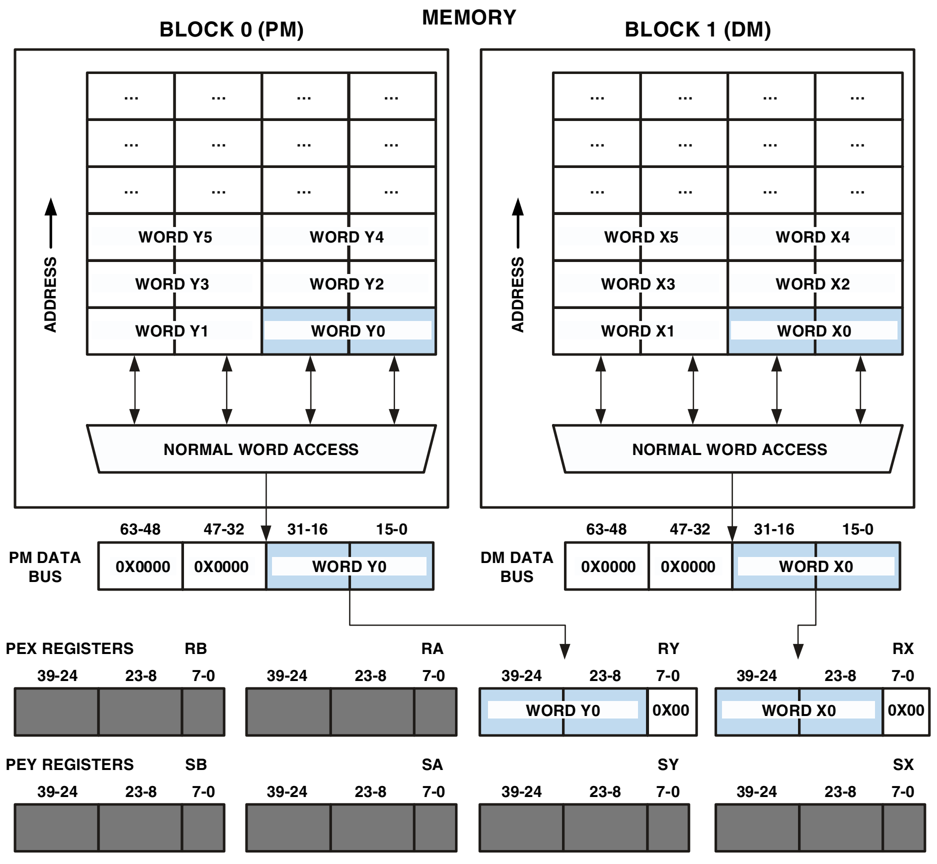
Figure 5-16. Normal Word Addressing of Single-Data in SIMD Mode

32-Bit Normal Word Addressing of Dual-Data in SISD Mode

Figure 5-17 shows the SISD dual-data, 32-bit normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The 32-bit values for normal word accesses transfer using the least significant normal word lanes of the PM and DM data buses. The processor drives the other normal word lanes of the data buses with zeros. Note that the accesses on both buses do not have to be the same word width. SISD mode dual-data accesses can handle any combination of short word, normal word, extended-precision normal word, or long word accesses. For more information, see "Mixed-Word Width Addressing of Dual-Data in SISD Mode" on page 5-68.

In Figure 5-17, the access targets the `PEx` registers in a SISD mode operation. This instruction accesses `WORD X0` in block 1 and `WORD Y0` in block 0. Each of these words has a normal word address with 0 for its least significant address bit. Other accesses within these four column locations have addresses with the least significant bit of 1 and select `WORD X1/Y1` from memory. The syntax targets registers `RX` and `RY` in `PEx`. The example targets `PEy` registers when using the syntax `SX` or `SY`.

For normal word accesses, the processor zero-fills the least significant 8 bits of the data register on loads and truncates these bits on stores to memory.



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS), RY = PM(NORMAL WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, NORMAL WORD, DUAL-DATA TRANSFERS ARE:

| DREG = PM(NORMAL WORD ADDRESS), | DREG = DM(NORMAL WORD ADDRESS); |
| PM(NORMAL WORD ADDRESS) = DREG, | DM(NORMAL WORD ADDRESS) = DREG; |

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-17. Normal Word Addressing of Dual-Data in SISD Mode

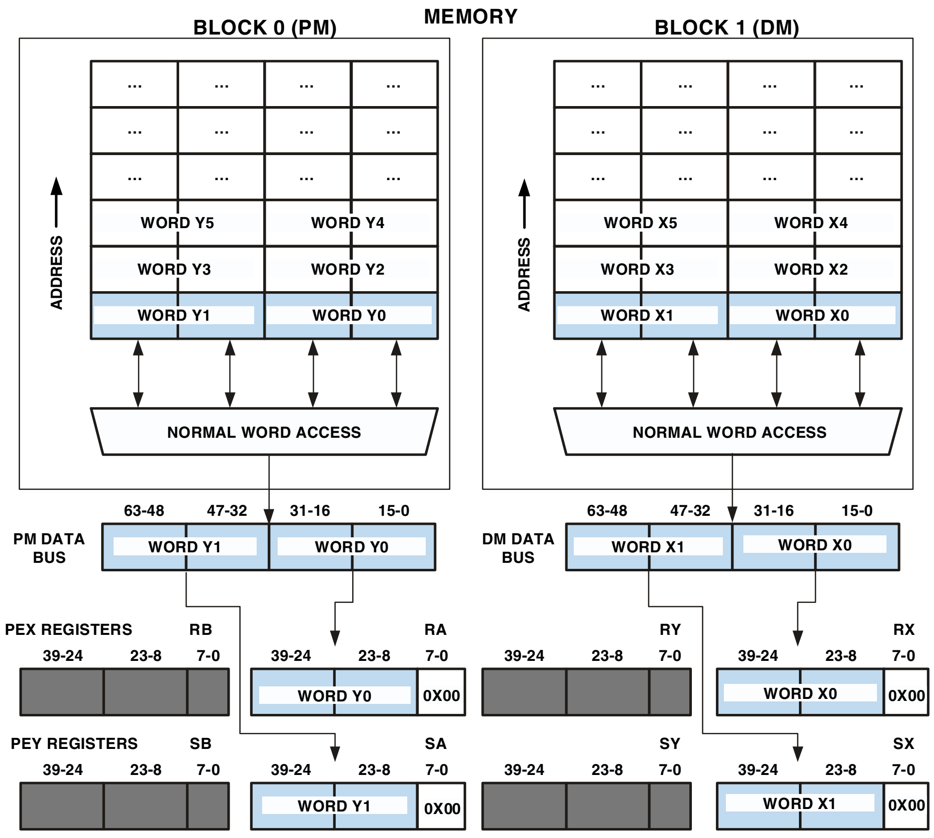
32-Bit Normal Word Addressing of Dual-Data in SIMD Mode

Figure 5-18 shows the SIMD, dual-data, 32-bit normal word addressed access mode. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The explicitly addressed (named in the instruction) 32-bit values are transferred using the least significant normal word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) normal word values are transferred using the most significant normal word lanes of the PM and DM data bus. Note that the accesses on both buses do not have to be the same word width. SIMD mode dual-data accesses can handle combinations of short word and normal word or extended-precision normal word and long word accesses. For more information, see "Mixed-Word Width Addressing of Dual-Data in SIMD Mode" on page 5-70.

In Figure 5-18, the explicit access targets the named registers `RX` and `RA`, and the implicit access targets those register's complementary registers `SX` and `SA`. This instruction uses the `PEx` registers with the `RX` and `RA` mnemonics. If the syntax named `PEy` registers `SX` and `SA` as the explicit targets, the processor would use those registers' complements, `RX` and `RA`, as the implicit targets. For more information on complementary registers, see "Secondary Processing Element (PEy)" on page 2-45.

For normal word accesses, the processor zero-fills the least significant 8 bits of the data register on loads and truncates these bits on stores to memory.

Figure 5-17 shows the data path for one transfer. The processor accesses normal words sequentially in memory as shown in Table 5-7 on page 5-50. For more information on arranging data in memory to take advantage of this access pattern, see Figure 5-34 on page 5-80.



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS), RY = PM(NORMAL WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, NORMAL WORD, DUAL-DATA TRANSFERS ARE:

|DREG = PM(NORMAL WORD ADDRESS), | DREG = DM(NORMAL WORD ADDRESS);|
|PM(NORMAL WORD ADDRESS) = DREG, | DM(NORMAL WORD ADDRESS) = DREG;|

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-18. Normal Word Addressing of Dual-Data in SIMD Mode

Extended-Precision Normal Word Addressing of Single-Data

Figure 5-19 on page 5-57 displays a possible single-data, 40-bit extended-precision normal word addressed access. For extended-precision normal word addressing, the processor treats each data bus as a 40-bit extended-precision normal word lane. The 40-bit value for the extended-precision normal word access is transferred using the most significant 40 bits of the PM or DM data bus. The processor drives the lower 24 bits of the data buses with zeros.

In Figure 5-19, the access targets a `PEx` register in a SISD or SIMD mode operation; extended-precision normal word single-data access operate the same in SISD or SIMD mode. This instruction accesses `WORD X0` with syntax that targets register `RX` in `PEx`. The example targets a `PEy` register when using the syntax `SX`.

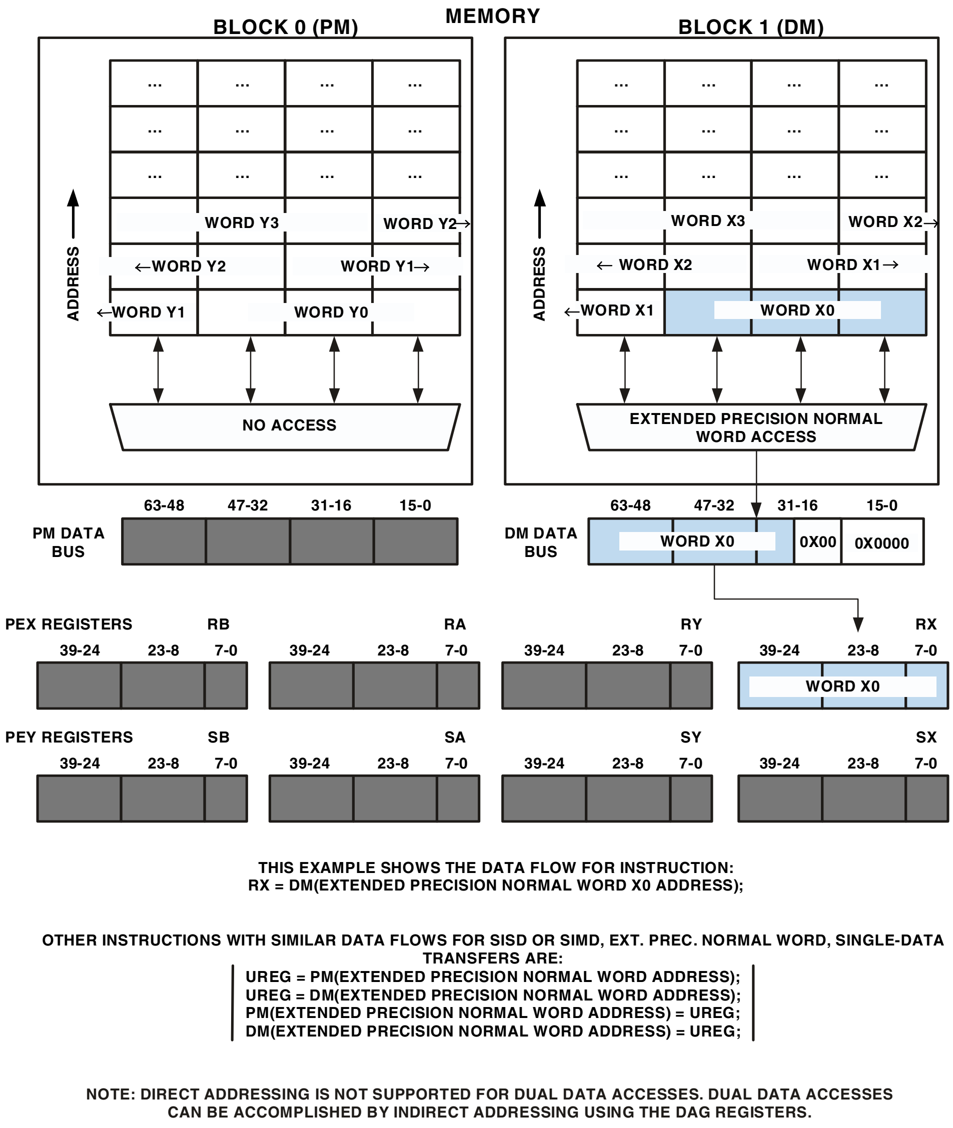
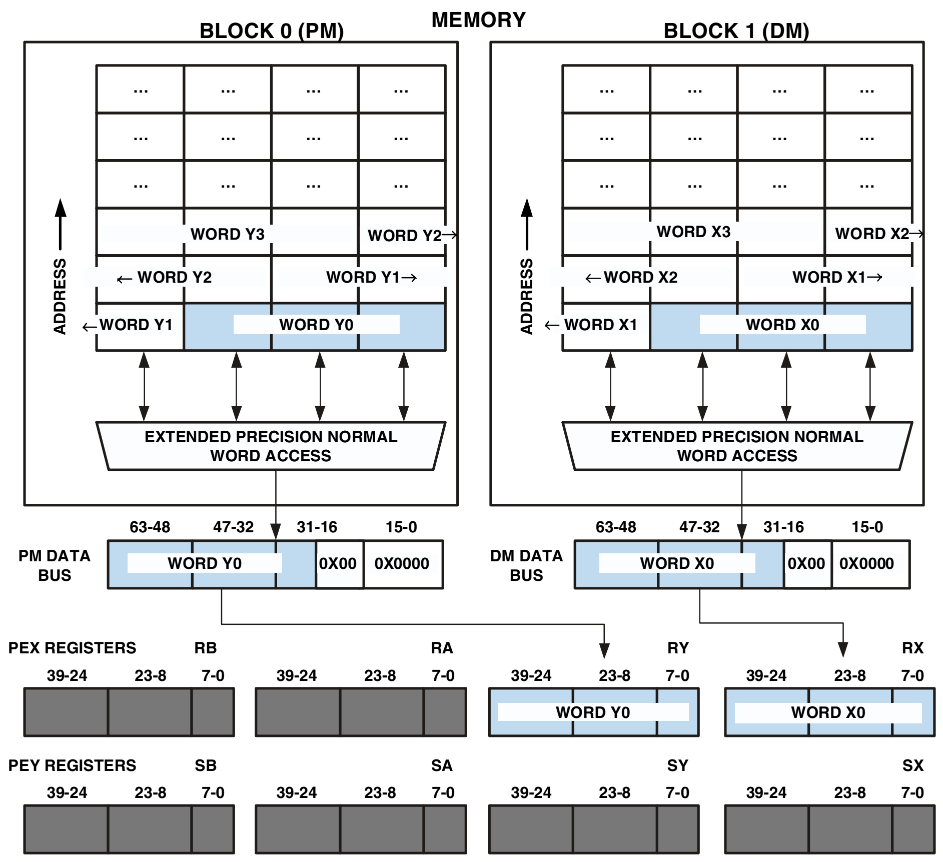

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y3
WORD Y2
WORD Y2
WORD Y1
WORD Y1
WORD Y0
NO ACCESS
WORD X3
WORD X2
WORD X2
WORD X1
WORD X1
WORD X0
EXTENDED PRECISION NORMAL WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
WORD X0
0X00
0X0000
PEX REGISTERS
RB
RA
RY
RX
PEY REGISTERS
SB
SA
SY
SX
39-24
23-8
7-0
WORD X0
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(EXTENDED PRECISION NORMAL WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD OR SIMD, EXT. PREC. NORMAL WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(EXTENDED PRECISION NORMAL WORD ADDRESS);
UREG = DM(EXTENDED PRECISION NORMAL WORD ADDRESS);
PM(EXTENDED PRECISION NORMAL WORD ADDRESS) = UREG;
DM(EXTENDED PRECISION NORMAL WORD ADDRESS) = UREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-19. Extended-Precision Normal Word Addressing of Single-Data

Extended-Precision Normal Word Addressing of Dual-Data in SISD Mode

Figure 5-20 shows the SISD, dual-data, 40-bit extended-precision normal word addressed access mode. For extended-precision normal word addressing, the processor treats each data bus as a 40-bit extended-precision normal word lane. The 40-bit values for the extended-precision normal word accesses are transferred using the most significant 40 bits of the PM and DM data bus. The processor drives the lower 24 bits of the data buses with zeros. Note that the accesses on both buses do not have to be the same word width. SISD mode, dual-data accesses can handle any combination of short word, normal word, extended-precision normal word, or long word accesses. For more information, see “Mixed-Word Width Addressing of Dual-Data in SISD Mode” on page 5-68.

In Figure 5-20, the access targets the `PEx` registers in a SISD mode operation. This instruction accesses `WORD X0` in block 1 and `WORD Y0` in block 0 with syntax that targets registers `RX` and `RY` in `PEx`. The example targets a `PEy` register when using the syntax `SX` or `SY`.



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(EP NORMAL WORD X0 ADDR.), RY = PM(EP NORMAL WORD Y0 ADDR.);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, EXTENDED PRECISION NORMAL WORD, DUAL-DATA TRANSFERS ARE:

DREG = PM(EXT. PREC. NORMAL WORD ADDRESS),	DREG = DM(EXT. PREC. NORMAL WORD ADDRESS);
PM(EXT. PREC. NORMAL WORD ADDRESS) = DREG,	DM(EXT. PREC. NORMAL WORD ADDRESS) = DREG;

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-20. Extended-Precision Normal Word Addressing of Dual-Data in SISD Mode

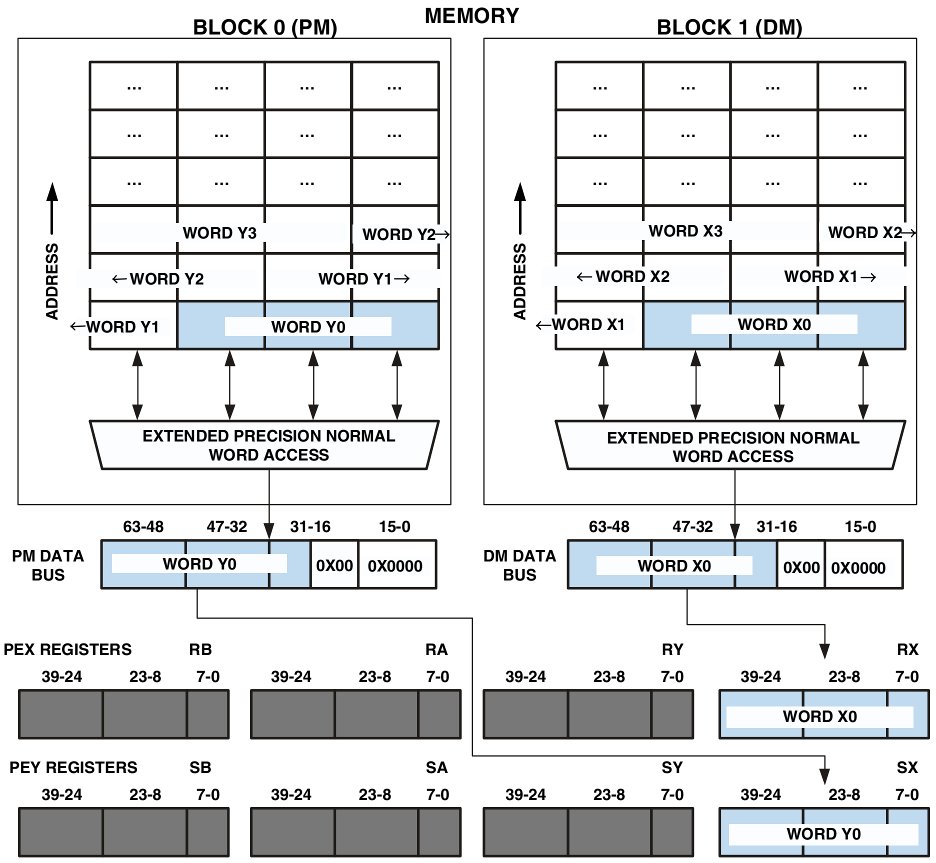
Extended-Precision Normal Word Addressing of Dual-Data in SIMD Mode

Figure 5-21 shows the SIMD, dual-data, 40-bit extended-precision normal word addressed access mode. For extended-precision normal word addressing, the processor treats each data bus as a 40-bit extended-precision normal word lane.

Because this word size approaches the limit of the data buses capacity, this SIMD mode transfer only moves the explicitly addressed locations and restricts data bus usage. The explicitly addressed (named in the instruction) 40-bit values that are transferred over the DM bus must source or sink a `PEx` data register, and the explicitly addressed (named in the instruction) 40-bit values that are transferred over the PM bus must source or sink a `PEy` data register; there are no implicit transfers in this mode. The 40-bit values for the extended-precision normal word accesses are transferred using the most significant 40 bits of the PM and DM data bus. The processor drives the lower 24 bits of the data buses with zeros.

The accesses on both buses do not have to be the same word width. This special case of SIMD mode dual-data accesses can handle any combination of extended-precision normal word or long word accesses. For more information, see "Mixed-Word Width Addressing of Dual-Data in SIMD Mode" on page 5-70.

In Figure 5-21, the access targets `PEx` and `PEy` registers in a SIMD mode operation. This instruction accesses `WORD X0` in block 1 with syntax that targets register `RX` in `PEx` and accesses `WORD Y0` in block 0 with syntax that targets register `SX` in `PEy`.



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(EP NORMAL WORD X0 ADDR.), SX = PM(EP NORMAL WORD Y0 ADDR.);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, EXTENDED PRECISION NORMAL WORD, DUAL-DATA TRANSFERS ARE:

PEY DREG = PM(EP NORMAL WORD ADDRESS),	PEX DREG = DM(EP NORMAL WORD ADDRESS);
PM(EP NORMAL WORD ADDRESS) = PEY DREG,	DM(EP NORMAL WORD ADDRESS) = PEX DREG;

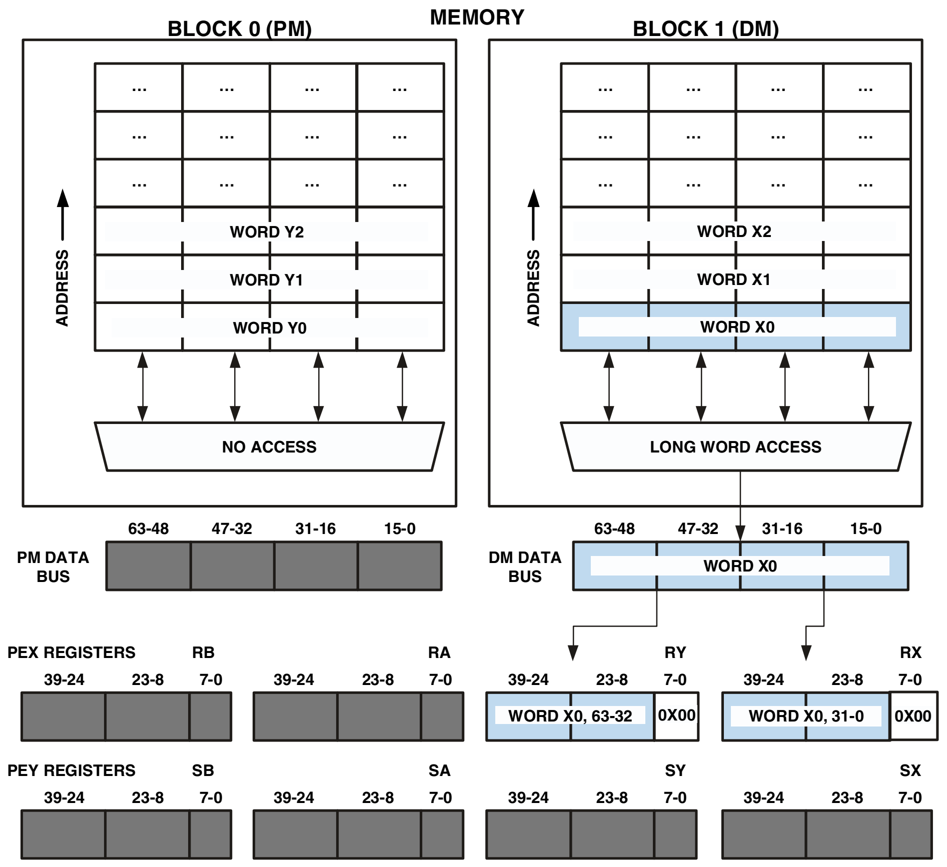
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-21. Extended-Precision Normal Word Addressing of Dual-Data in SIMD Mode

Long Word Addressing of Single-Data

Figure 5-22 displays one possible single-data, long word addressed access. For long word addressing, the processor treats each data bus as a 64-bit long word lane. The 64-bit value for the long word access completes a transfer using the full width of the PM or DM data bus.

In Figure 5-22, the access targets a `PEx` register in a SISD or SIMD mode operation. Long word single-data access operate the same in SISD or SIMD mode. This instruction accesses `WORD X0` with syntax that explicitly targets register `RX` and implicitly targets its neighbor register, `RY`, in `PEx`. The processor zero-fills the least significant 8 bits of both the registers. The example targets `PEy` registers when using the syntax `SX`. For more information on how neighbor registers (listed in Table 5-6) work, see "Long Word (64-Bit) Accesses" on page 5-32.



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD OR SIMD, LONG WORD, SINGLE-DATA TRANSFERS ARE:

```
UREG = PM(LONG WORD ADDRESS);
UREG = DM(LONG WORD ADDRESS);
PM(LONG WORD ADDRESS) = UREG;
DM(LONG WORD ADDRESS) = UREG;
```

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-22. Long Word Addressing of Single-Data

Long Word Addressing of Dual-Data in SISD Mode

Figure 5-23 shows the SISD, dual-data, long word addressed access mode. For long word addressing, the processor treats each data bus as a 64-bit long word lane. The 64-bit values for the long word accesses completes a transfer using the full width of the PM or DM data bus.

In Figure 5-23, the access targets `PEx` registers in SISD mode operation. This instruction accesses `WORD X0` and `WORD Y0` with syntax that explicitly targets registers `RX` and `RA` and implicitly targets their neighbor registers `RY` and `RB` in `PEx`. The processor zero-fills the least significant 8 bits of all the registers. The example targets `PEy` registers when using the syntax `SX` and `SA`. For more information on how neighbor registers (listed in Table 5-6) work, see "Long Word (64-Bit) Accesses" on page 5-32.

Programs must be careful not to explicitly target neighbor registers in this instruction. While the syntax lets programs target these registers, one of the explicit accesses targets the implicit target of the other access. The processor resolves this conflict by performing only the access with higher priority. For more information on the priority order of data register file accesses, see "Data Register File" on page 2-37.

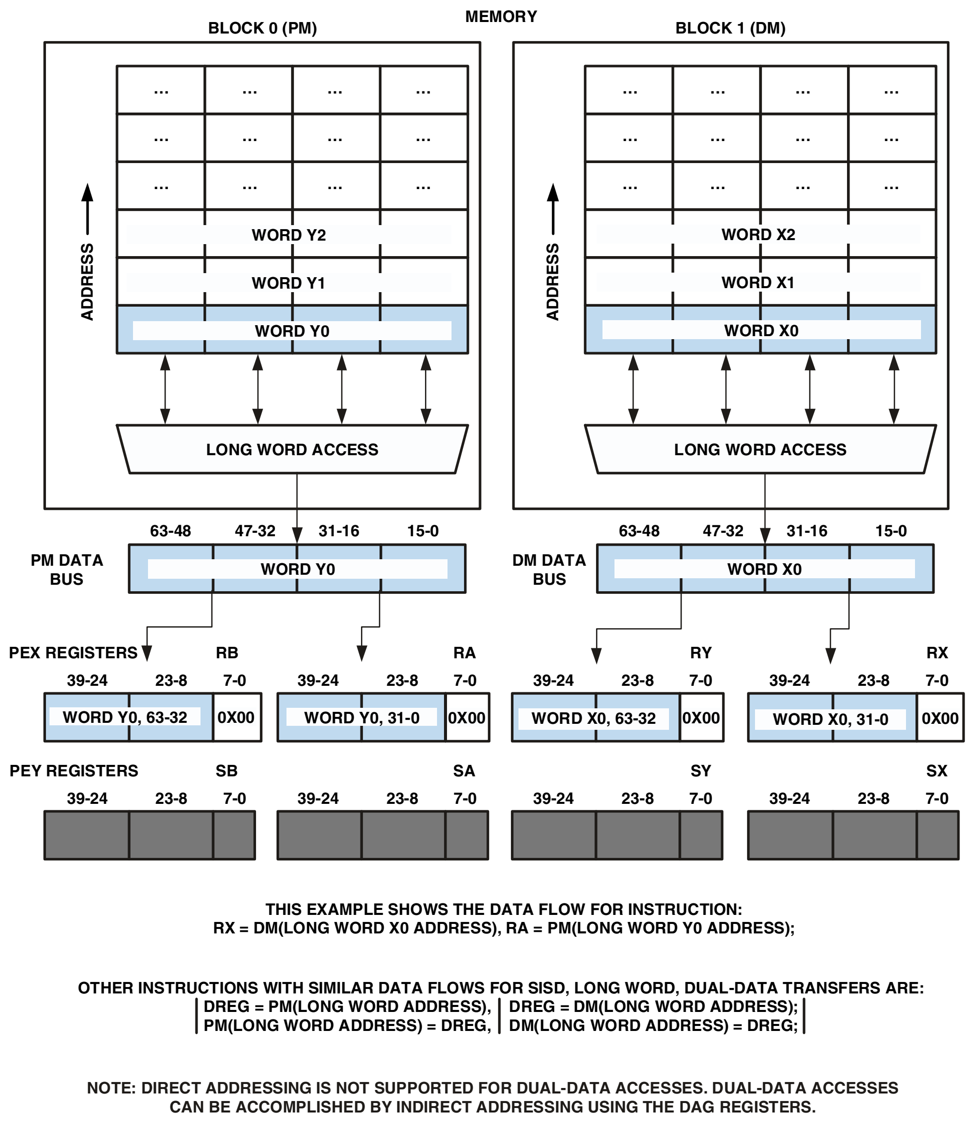

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y2
WORD Y1
WORD Y0
WORD X2
WORD X1
WORD X0
LONG WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
WORD Y0, 63-32
WORD Y0, 31-0
WORD X0, 63-32
WORD X0, 31-0
0X00
PEY REGISTERS
SB
SA
SY
SX
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS), RA = PM(LONG WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, LONG WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(LONG WORD ADDRESS), | DREG = DM(LONG WORD ADDRESS);
PM(LONG WORD ADDRESS) = DREG, | DM(LONG WORD ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-23. Long Word Addressing of Dual-Data in SISD Mode

Long Word Addressing of Dual-Data in SIMD Mode

Figure 5-24 shows the SIMD, dual-data, long word addressed access mode that targets internal memory space. For long word addressing, the processor treats each data bus as a 64-bit long word lane. The 64-bit values for the long word accesses completes a transfer using the full width of the PM or DM data bus.

Because this word size approaches the limit of the data buses' capacity, this SIMD mode transfer only moves the explicitly addressed locations and restricts data bus usage. The explicitly addressed (named in the instruction) 64-bit values transferred over the DM bus must source or sink a `PEx` data register, and the explicitly addressed (named in the instruction) 64-bit values transferred over the PM bus must source or sink a `PEy` data register; there are no implicit transfers in this mode.

In Figure 5-24, the access targets `PEx` and `PEy` registers in a SIMD mode operation. This instruction accesses `WORD X0` in block 1 with syntax that targets register `RX` and its neighbor register `RY` in `PEx` and accesses `WORD Y0` in block 0 with syntax that targets register `SX` and its neighbor register `SY` in `PEy`. The processor zero-fills the least significant 8 bits of all the registers. For more information on how neighbor registers (listed in Table 5-6) work, see "Long Word (64-Bit) Accesses" on page 5-32.

ⓘ The accesses on both buses do not have to be the same word width. This special case of SIMD mode dual-data accesses can handle any combination of extended-precision normal word or long word accesses. "Mixed-Word Width Addressing of Dual-Data in SIMD Mode" on page 5-70.

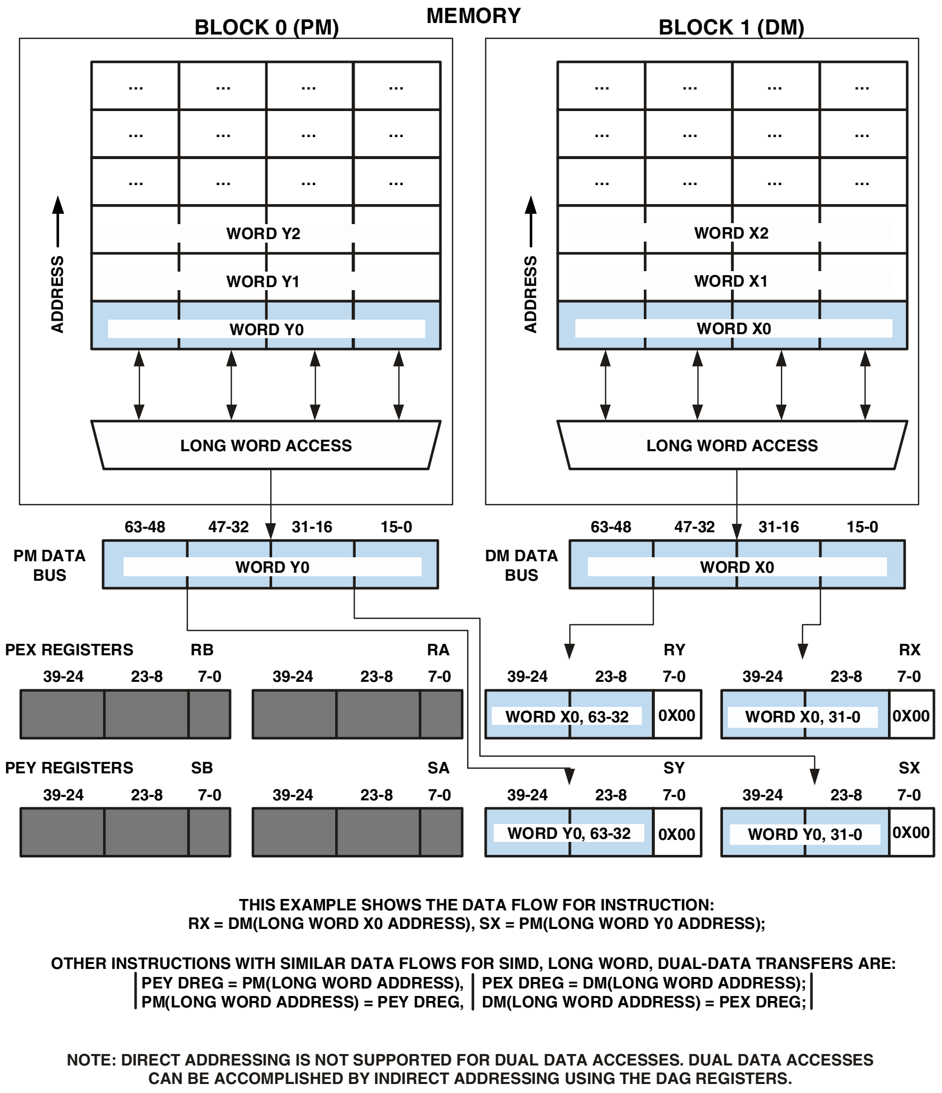
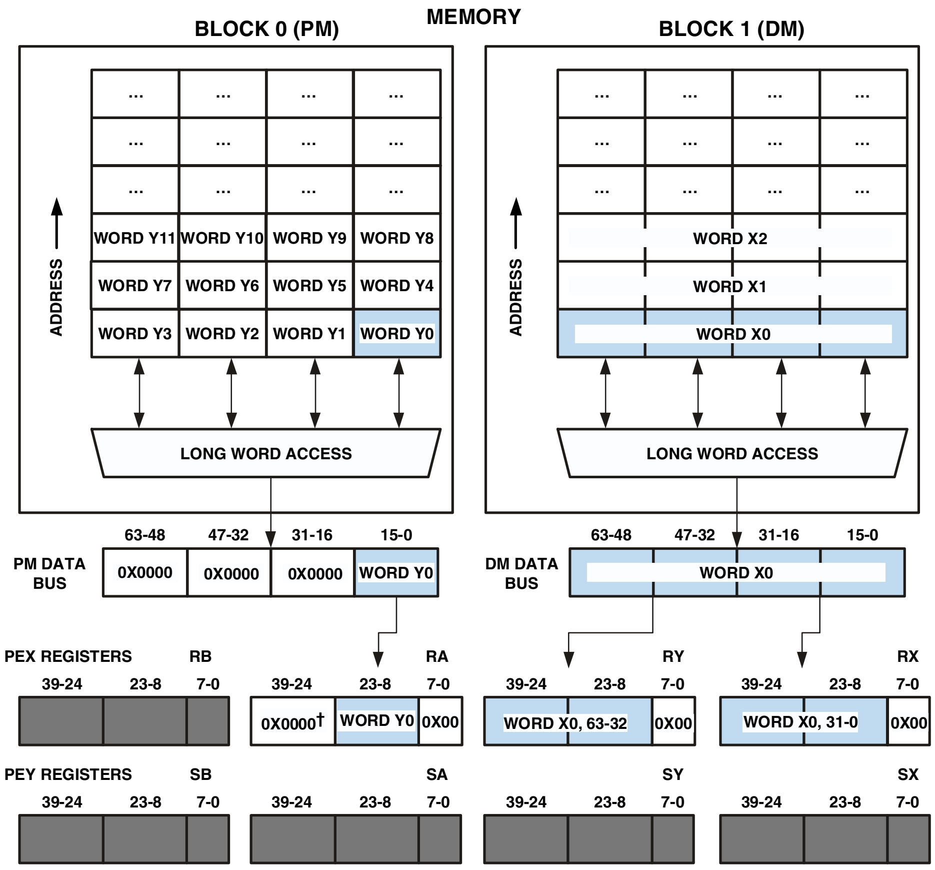

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
...
WORD Y2
WORD Y1
WORD Y0
WORD X2
WORD X1
WORD X0
LONG WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
WORD X0, 63-32
WORD X0, 31-0
0X00
PEY REGISTERS
SB
SA
SY
SX
WORD Y0, 63-32
WORD Y0, 31-0
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS), SX = PM(LONG WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, LONG WORD, DUAL-DATA TRANSFERS ARE:
PEY DREG = PM(LONG WORD ADDRESS),
PM(LONG WORD ADDRESS) = PEY DREG,
PEX DREG = DM(LONG WORD ADDRESS);
DM(LONG WORD ADDRESS) = PEX DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-24. Long Word Addressing of Dual-Data in SIMD Mode

Mixed-Word Width Addressing of Dual-Data in SISD Mode

Figure 5-25 shows an example of a mixed-word width, dual-data, SISD mode access. This example shows how the processor transfers a long word access on the DM bus and transfers a short word access on the PM bus. The memory architecture permits mixing all other combinations of dual-data SISD mode short word, normal word, extended-precision normal word, and long word accesses.

ⓘ In case of conflicting dual access to the data register file, the processor only performs the access with higher priority. For more information on how the processor prioritizes accesses, see "Data Register File" on page 2-37.



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS), RA = PM(SHORT WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, MIXED WORD, DUAL-DATA TRANSFERS ARE:

DREG = PM(SHORT, NORMAL, EP NORMAL, LONG ADD),	DREG = DM(SHORT, NORMAL, EP NORMAL, LONG ADD);
PM(SHORT, NORMAL, EP NORMAL, LONG ADD) = DREG,	DM(SHORT, NORMAL, EP NORMAL, LONG ADD) = DREG;

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-25. Mixed-Word Width Addressing of Dual-Data in SISD Mode

Mixed-Word Width Addressing of Dual-Data in SIMD Mode

Figure 5-26 shows an example of a mixed-word width, dual-data, SIMD mode access. This example shows how the processor transfers a long word access on the DM bus and transfers an extended-precision normal word access on the PM bus.

The memory architecture permits mixing SIMD mode dual-data short word and normal word accesses or extended-precision normal word and long word accesses. No other combinations of mixed word dual-data SIMD mode accesses are permissible.

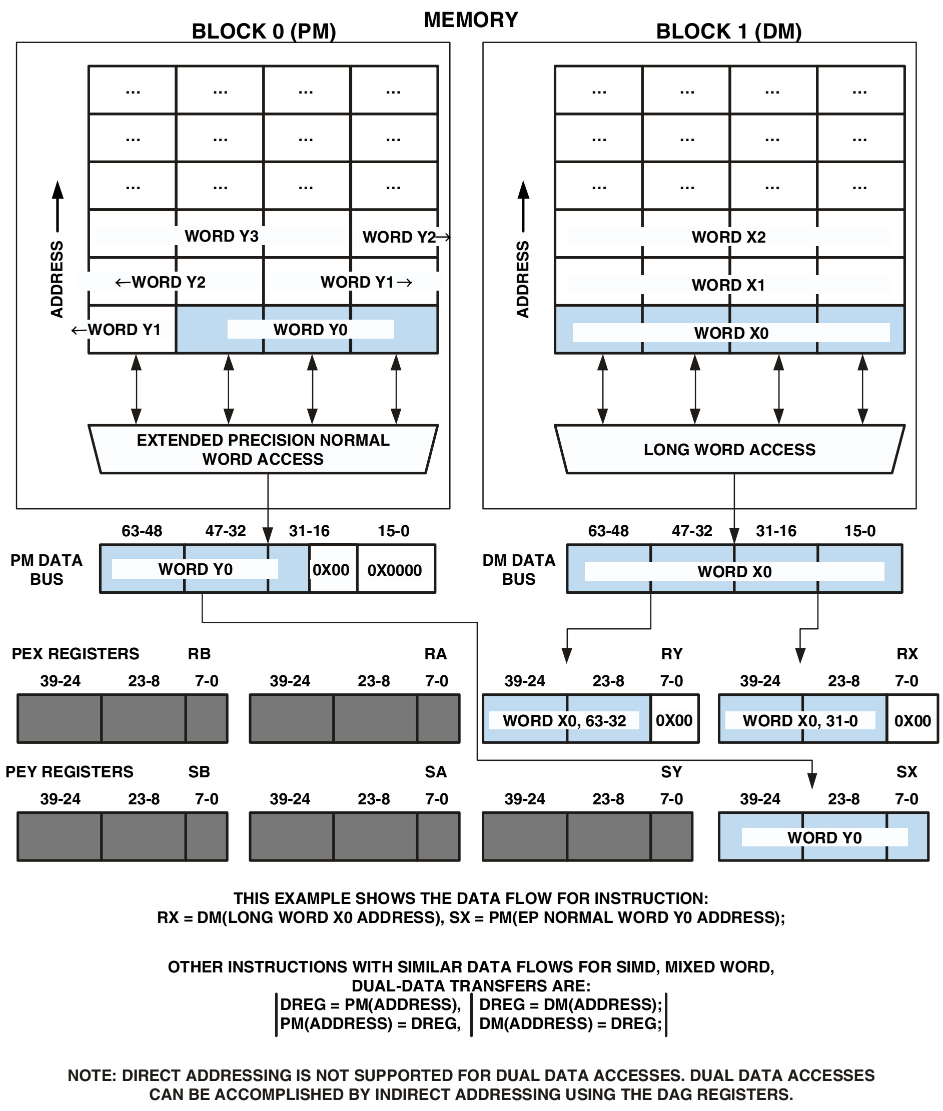

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y3
WORD Y2→
←WORD Y2
WORD Y1→
←WORD Y1
WORD Y0
WORD X2
WORD X1
WORD X0
EXTENDED PRECISION NORMAL WORD ACCESS
LONG WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
WORD Y0
0X00
0X0000
DM DATA BUS
WORD X0
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
WORD X0, 63-32
0X00
WORD X0, 31-0
0X00
PEY REGISTERS
SB
SA
SY
SX
WORD Y0
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS), SX = PM(EP NORMAL WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, MIXED WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(ADDRESS), DREG = DM(ADDRESS);
PM(ADDRESS) = DREG, DM(ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-26. Mixed-Word Width Addressing of Dual-Data in SIMD Mode

Broadcast Load Access

Figure 5-27 through Figure 5-34 provide examples of broadcast load accesses for single and dual-data transfers. These examples show that the broadcast load's memory and register access is a hybrid of the corresponding non-broadcast SISD and SIMD mode accesses. The exceptions to this relation are broadcast load dual-data, extended-precision normal word and long word accesses. These broadcast accesses differ from their corresponding non-broadcast mode accesses.

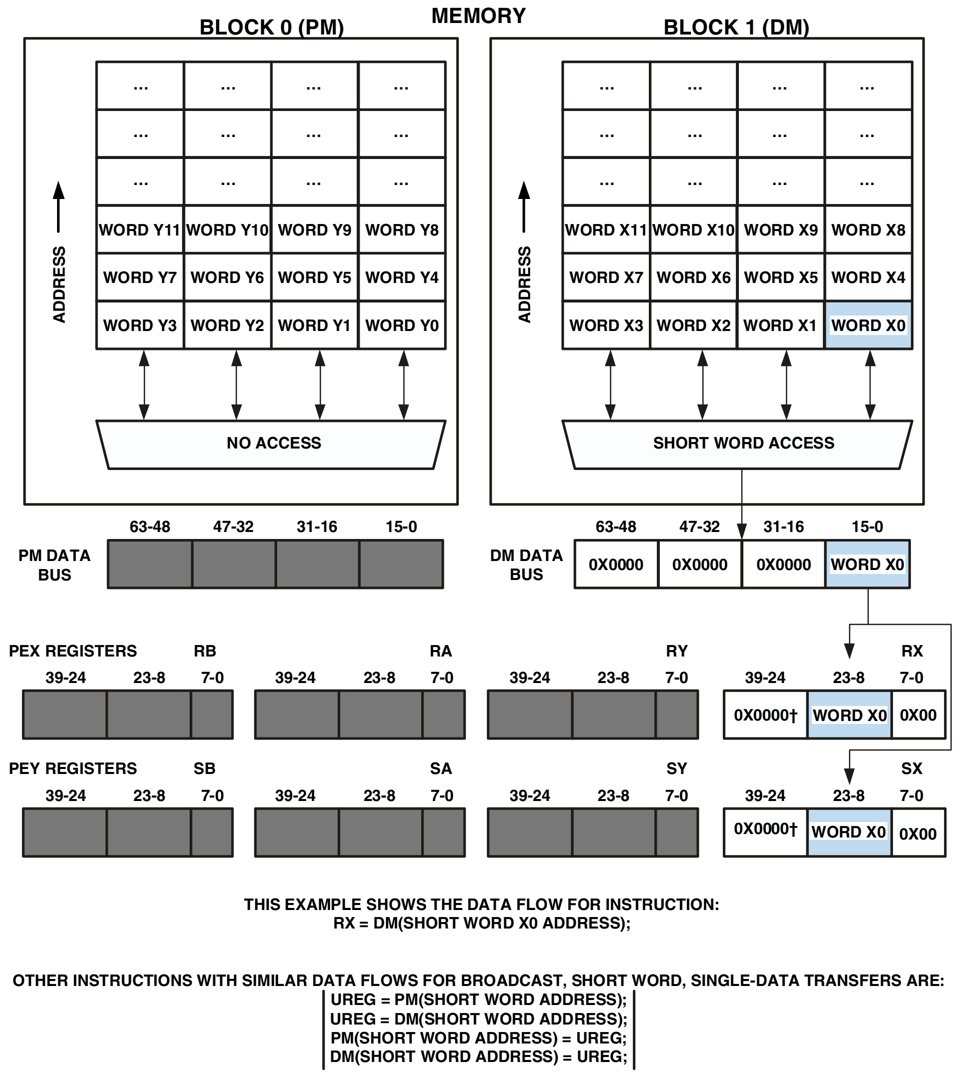


Figure 5-27. Short Word Addressing of Single-Data in Broadcast Load

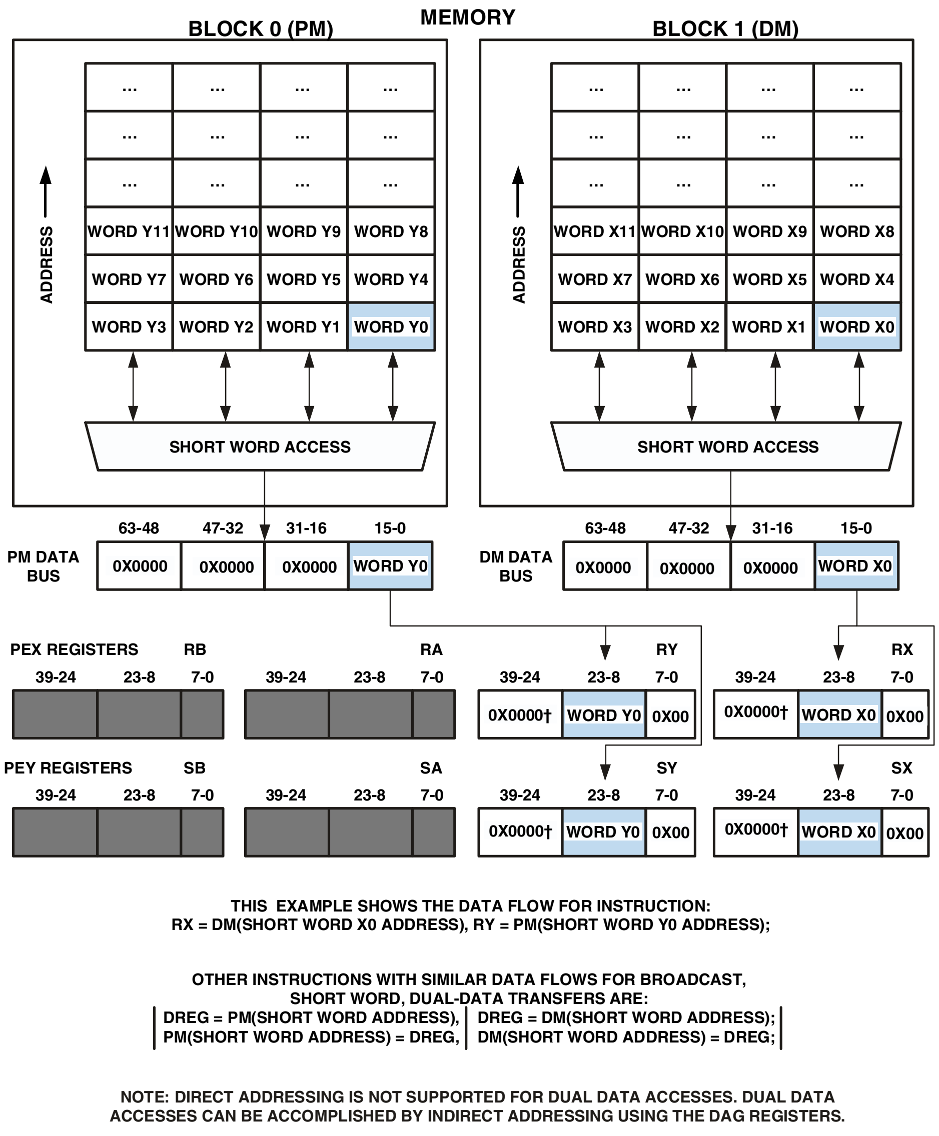
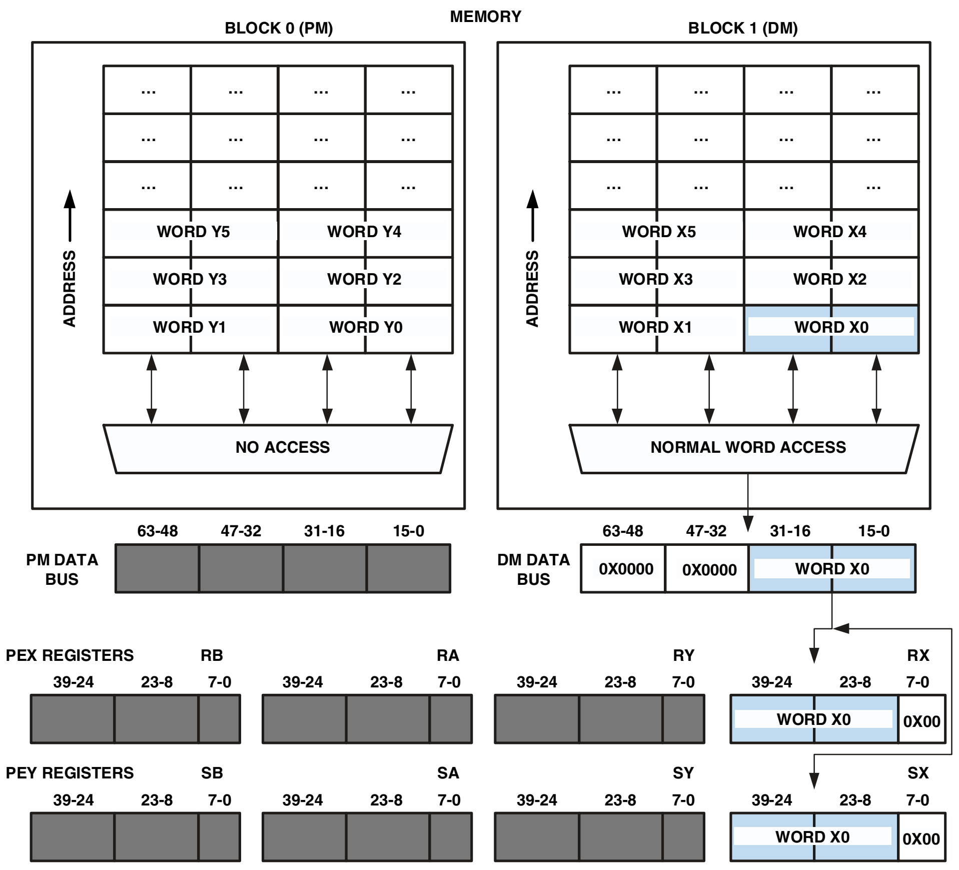

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
...
WORD Y11
WORD Y10
WORD Y9
WORD Y8
WORD Y7
WORD Y6
WORD Y5
WORD Y4
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X11
WORD X10
WORD X9
WORD X8
WORD X7
WORD X6
WORD X5
WORD X4
WORD X3
WORD X2
WORD X1
WORD X0
SHORT WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
0X0000
WORD Y0
DM DATA BUS
WORD X0
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
0X0000†
0X00
PEY REGISTERS
SB
SA
SY
SX
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(SHORT WORD X0 ADDRESS), RY = PM(SHORT WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, SHORT WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(SHORT WORD ADDRESS), | DREG = DM(SHORT WORD ADDRESS);
PM(SHORT WORD ADDRESS) = DREG, | DM(SHORT WORD ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-28. Short Word Addressing of Dual-Data in Broadcast Load



THE ABOVE EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, NORMAL WORD, SINGLE-DATA TRANSFERS ARE:

```
UREG = PM(NORMAL WORD ADDRESS);
UREG = DM(NORMAL WORD ADDRESS);
PM(NORMAL WORD ADDRESS) = UREG;
DM(NORMAL WORD ADDRESS) = UREG;
```

Figure 5-29. Normal Word Addressing of Single-Data in Broadcast Load

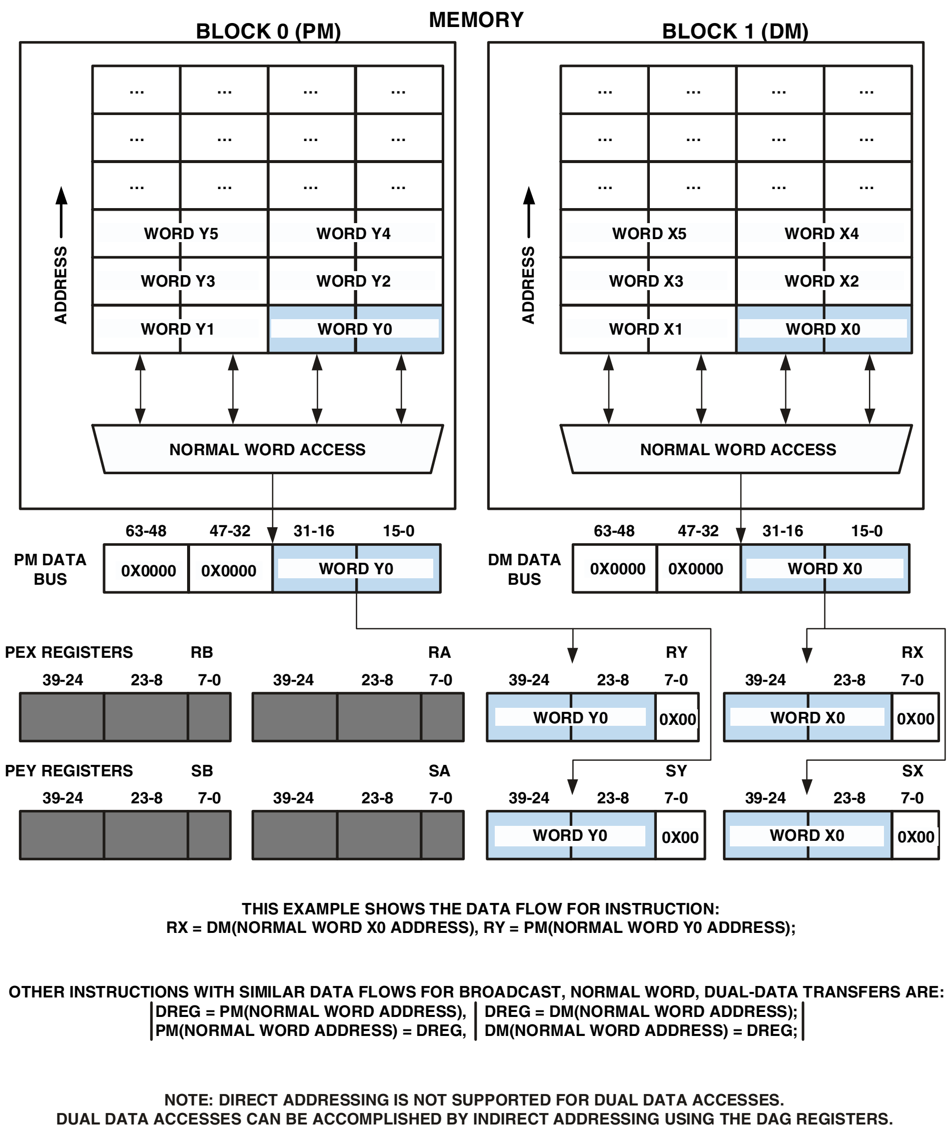

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
...
WORD Y5
WORD Y4
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X5
WORD X4
WORD X3
WORD X2
WORD X1
WORD X0
NORMAL WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
0X0000
PEX REGISTERS
PEY REGISTERS
RB
RA
RY
RX
SB
SA
SY
SX
39-24
23-8
7-0
0X00
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS), RY = PM(NORMAL WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, NORMAL WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(NORMAL WORD ADDRESS), | DREG = DM(NORMAL WORD ADDRESS);
PM(NORMAL WORD ADDRESS) = DREG, | DM(NORMAL WORD ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES.
DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-30. Normal Word Addressing of Dual-Data in Broadcast Load

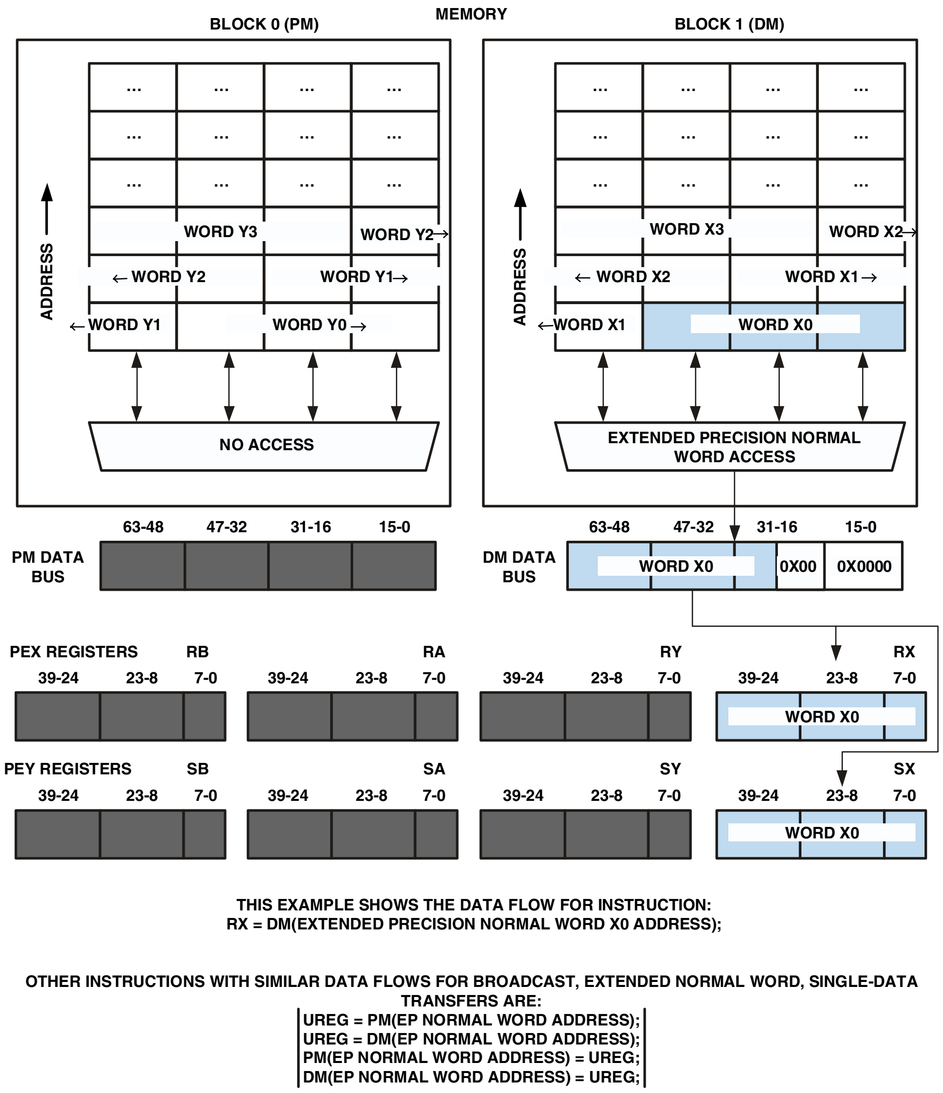

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X3
WORD X2
WORD X1
WORD X0
NO ACCESS
EXTENDED PRECISION NORMAL WORD ACCESS
PM DATA BUS
DM DATA BUS
63-48
47-32
31-16
15-0
0X00
0X0000
PEX REGISTERS
PEY REGISTERS
RB
RA
RY
RX
SB
SA
SY
SX
39-24
23-8
7-0
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(EXTENDED PRECISION NORMAL WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, EXTENDED NORMAL WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(EP NORMAL WORD ADDRESS);
UREG = DM(EP NORMAL WORD ADDRESS);
PM(EP NORMAL WORD ADDRESS) = UREG;
DM(EP NORMAL WORD ADDRESS) = UREG;

Figure 5-31. Extended-Precision Normal Word Addressing of Single-Data in Broadcast Load

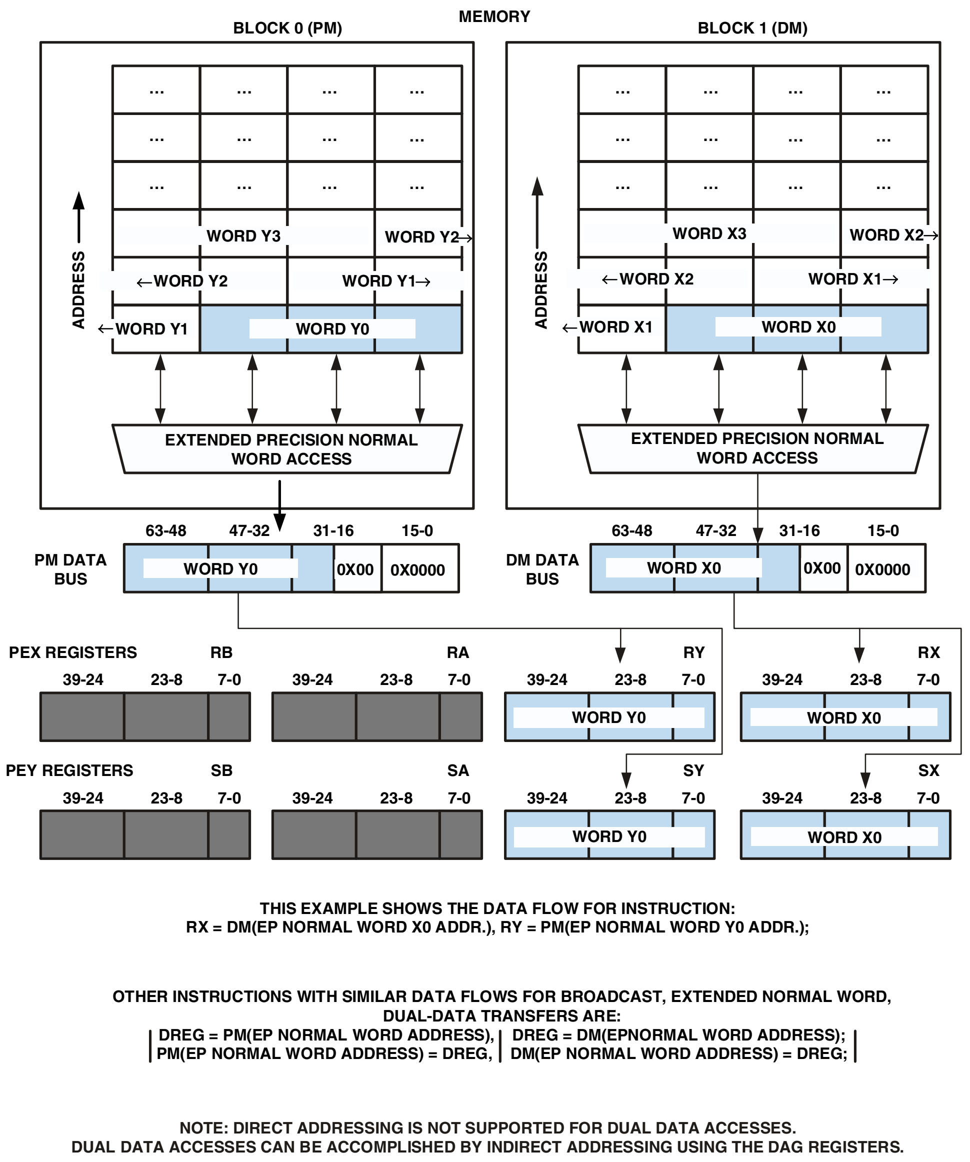

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
...
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X3
WORD X2
WORD X1
WORD X0
EXTENDED PRECISION NORMAL WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
0X00
0X0000
PEX REGISTERS
PEY REGISTERS
RB
RA
RY
RX
SB
SA
SY
SX
39-24
23-8
7-0
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(EP NORMAL WORD X0 ADDR.), RY = PM(EP NORMAL WORD Y0 ADDR.);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, EXTENDED NORMAL WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(EP NORMAL WORD ADDRESS), | DREG = DM(EPNORMAL WORD ADDRESS);
PM(EP NORMAL WORD ADDRESS) = DREG, | DM(EP NORMAL WORD ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES.
DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-32. Extended-Precision Normal Word Addressing of Dual-Data in Broadcast Load

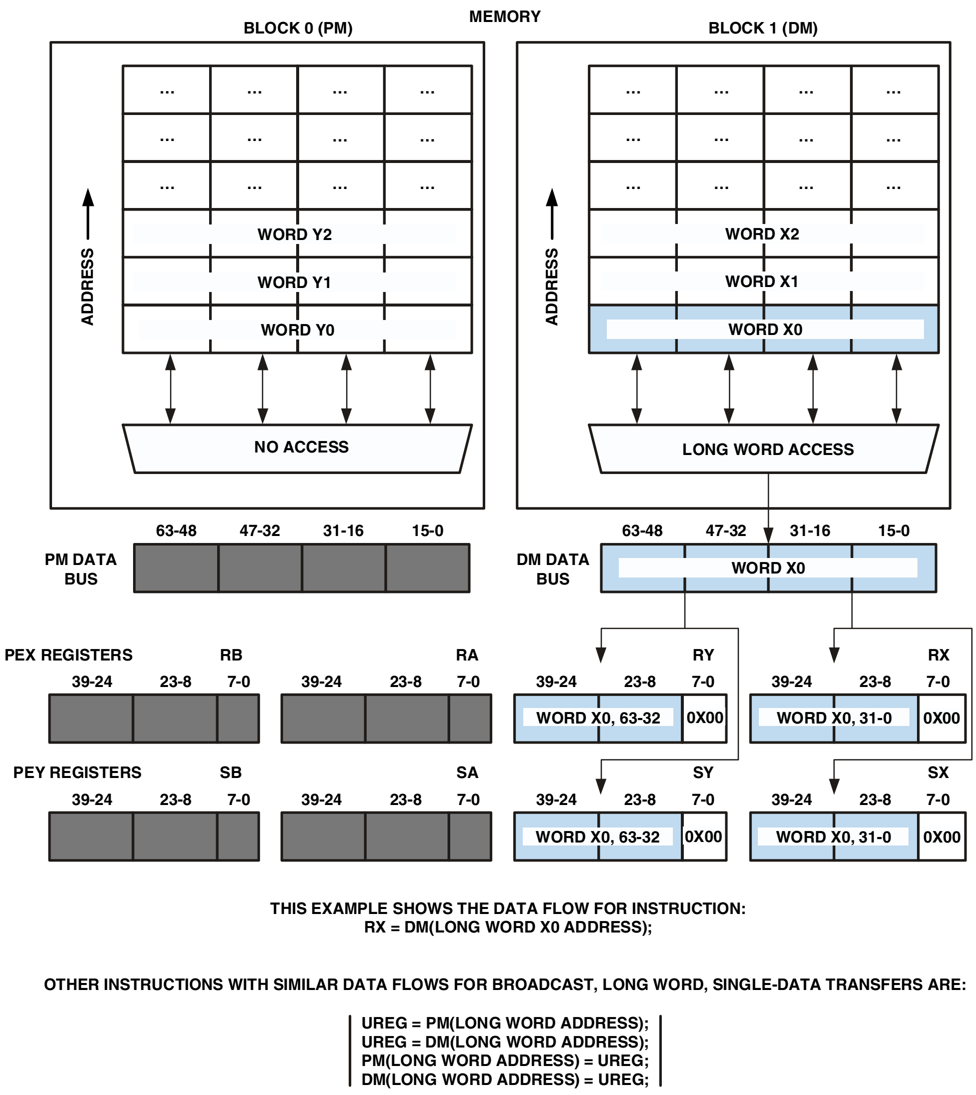
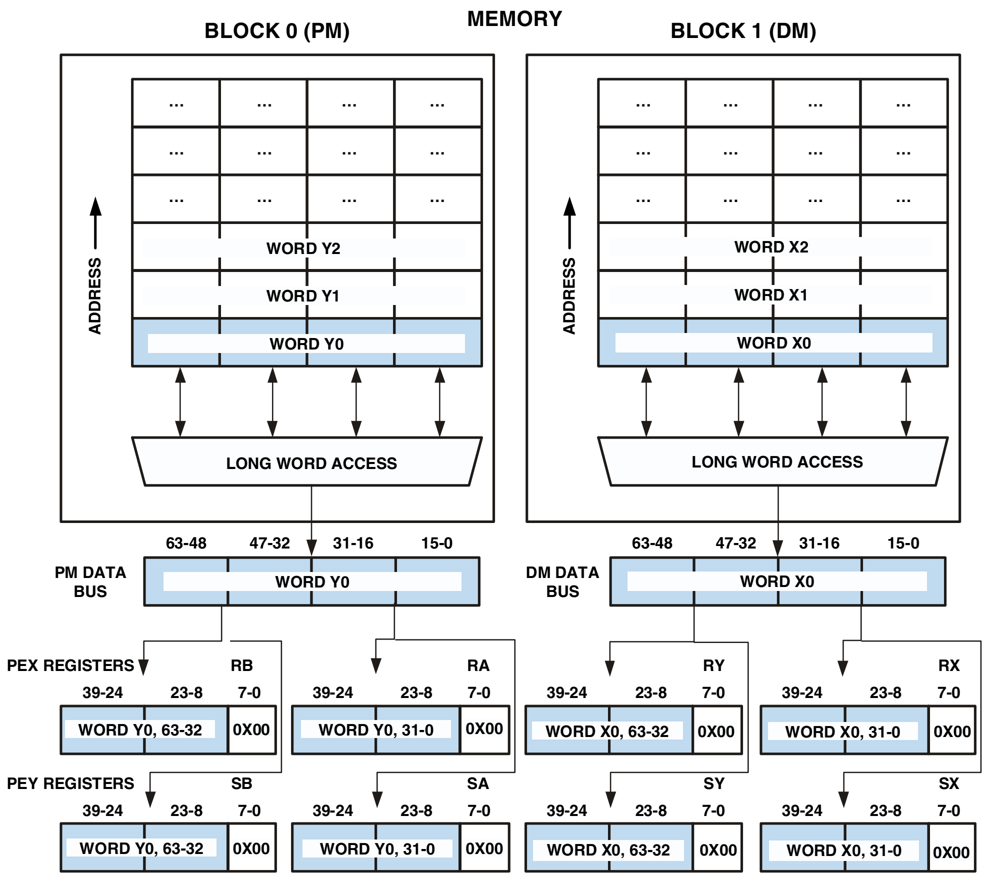

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
...
WORD Y2
WORD Y1
WORD Y0
WORD X2
WORD X1
WORD X0
NO ACCESS
LONG WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
WORD X0
PEX REGISTERS
RB
RA
RY
RX
PEY REGISTERS
SB
SA
SY
SX
39-24
23-8
7-0
WORD X0, 63-32
0X00
WORD X0, 31-0
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, LONG WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(LONG WORD ADDRESS);
UREG = DM(LONG WORD ADDRESS);
PM(LONG WORD ADDRESS) = UREG;
DM(LONG WORD ADDRESS) = UREG;

Figure 5-33. Long Word Addressing of Single-Data in Broadcast Load



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS), RA = PM(LONG WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, LONG WORD, DUAL-DATA TRANSFERS ARE:

| DREG = PM(LONG WORD ADDRESS), | DREG = DM(LONG WORD ADDRESS);|
| PM(LONG WORD ADDRESS) = DREG, | DM(LONG WORD ADDRESS) = DREG;|

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES.
DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-34. Long Word Addressing of Dual-Data in Broadcast Load

Shadow Write FIFO

Because the processor's internal memory operates at high speeds, writes to the memory do not go directly into the memory array, but rather to a two-deep FIFO called the shadow write FIFO. This FIFO uses a non-read cycle (either a write cycle, or a cycle in which there is no access of internal memory) to load data from the FIFO into internal memory. When an internal memory write cycle occurs, the FIFO loads any data from a previous write into memory and accepts new data.

Shadow Write FIFO Use in SIMD Mode

The shadow write FIFO is located between the internal memory array of the ADSP-2136x processor and the core.

When performing SIMD reads that cross long word address boundaries and the data read resides in the shadow write FIFO, the read in SIMD mode causes unpredictable results for explicit accesses of odd normal word addresses in internal memory. The implicit part of this SIMD mode transfer incorrectly accesses the previous sequential even address when the data is in the shadow write FIFO.

When the read data resides in internal memory, a SIMD mode explicit access to normal word address 0x98001 will result in an implicit access to the next sequential even address value. As shown in Table 5-8, a SIMD mode explicit access to normal word address 0x98001 results in an implicit access to normal word address 0x98002.

Table 5-8. Data Resides in Internal Memory

	Explicit "R0" R0 = dm(I0,M0)		**Explicit "S0" S0 = dm(I0,M0);**	
Explicit Address (I0)	R0	S0	R0	S0
0x80001	32-bit word at 0x80001	32-bit word at 0x80002	32-bit word at 0x80002	32-bit word at 0x80001

6 JTAG TEST EMULATION PORT

In addition to boundary scan, the JTAG test emulation port supports other functions including background telemetry channels, cycle counting with EMUCLK, user-configurable hardware breakpoint support, breakpoints, and a register for viewing the revision ID.

JTAG Test Access Port

The emulator uses JTAG boundary scan logic for ADSP-2136x processor communications and control. This JTAG logic consists of a state machine, a five pin test access port (TAP), and shift registers. The state machine and pins conform to the IEEE 1149.1 specification. The TAP pins appear in Table 6-1. A special pin ($\overline{\text{EMU}}$) is used in the JTAG emulators from Analog Devices. This pin is **not** defined in the IEEE-1149.1 specification. This signal notifies the ADSP-21364 EZ-KIT Lite that the processor has completed an operation.

Table 6-1. JTAG Test Access Port (TAP) Pins

Pin	I/O	Function
TCK	I	Test Clock: pin used to clock the TAP state machine (Asynchronous with CLKIN)
TMS	I	Test Mode Select: pin used to control the TAP state machine sequence
TDI	I	Test Data In: serial shift data input pin
TDO	O	Test Data Out: serial shift data output pin
$\overline{\text{TRST}}$	I	Test Logic Reset: resets the TAP state machine

A boundary scan description language (BSDL) file for the ADSP-2136x processor is available on the Analog Devices Web site.

Refer to the IEEE 1149.1 JTAG specification for detailed information on the JTAG interface. This chapter assumes a working knowledge of the JTAG specification.

Boundary Scan

A boundary scan allows a system designer to test interconnections on a printed circuit board with minimal test-specific hardware. The scan is made possible by the ability to control and monitor each input and output pin on each chip through a set of serially scannable latches. Each input and output is connected to a latch, and the latches are connected as a long shift register so that data can be read from or written to them through a serial test access port (TAP). The ADSP-2136x processor contains a test access port compatible with the industry-standard IEEE 1149.1 (JTAG) specification. Only the IEEE 1149.1 features specific to the ADSP-2136x are described here. For more information, see the IEEE 1149.1 specification and the other documents listed in "References" on page 6-17.

The boundary scan allows a variety of functions to be performed on each input and output signal of the ADSP-2136x processor. Each input has a latch that monitors the value of the incoming signal and can also drive data into the chip in place of the incoming value. Similarly, each output has a latch that monitors the outgoing signal and can also drive the output in place of the outgoing value. For bidirectional pins, the combination of input and output functions is available.

Every latch associated with a pin is part of a single serial shift register path. Each latch is a master/slave type latch with the controlling clock provided externally. This clock (`TCK`) is asynchronous to the ADSP-2136x system clock (`CLKIN`).

The processor emulation features halt the processor at a predefined point to examine the state of the processor, execute arbitrary code, restore the original state, and continue execution.

The ADSP-2136x processor emulation features are a superset of the ADSP-21160 processor emulation features. All emulation features supported by previous SHARC processors are supported on the ADSP-21364 processor.

There are several changes/extensions to the base functionality of the ADSP-2116x processor emulation capability, which require changes in the ADSP-21364 EZ-KIT software for ADSP-2136x processor support. These extensions include:

- New registers for added functionality: `EEMUCTL`, `EEMUSTAT`, `EEMUIN`, `EEMUOUT`, and `SHADOW_SHIFT`
- A new JTAG instruction to support these additional registers: `EEMUINDATA`, `EEMUOUTDATA`, and `EEMUCTL`
- New functionality to allow the tools software to support statistical profiling
- Support for background telemetry, user-definable breakpoint interrupts, and cycle counting

Several on-chip facilities are directly accessed through the JTAG interface. These facilities are listed in Table 6-2 on page 6-6. Other emulation facilities are only indirectly accessible. To indirectly access the facilities that do not appear in Table 6-2, scan the instruction which moves data of interest to/from the `PX` register, scan the `PX` data (if the instruction is a `PX` read), let the core execute the instruction, and then scan the `PX` register out (if the instruction is a `PX` write).

The breakpoint start/end registers are mapped into the IOP register space of the ADSP-2136x. The EMUN, EMUCLK, and EMUCLK2 registers occupy the same *Ureg* address space as the ADSP-2106x processor. These facilities are read-only by the ADSP-2136x processor core in normal operation.

Background Telemetry Channel (BTC)

Programmers can read and write data to a set of memory-mapped buffers (EEMUIN and EEMUOUT) that are accessible by the emulator while the core is running. This function allows the emulator to feed new data to the processor or get updates from the processor in real time. A 32-bit memory-mapped I/O register called EEMUSTAT can be used to enable this functionality and check the status of the input and output data buffers. Low priority emulator interrupts are generated when the EEMUIN buffer is full or the EEMUOUT FIFO is empty so that the processor core can handle reading/writing data from/to the buffers in an interrupt service routine (ISR). These interrupts are handled in the same way that normal interrupts are handled in the processor.

User-Definable Breakpoint Interrupts

Breakpoint interrupts enable users to write to the breakpoint registers directly so that they can induce an interrupt. Such interrupts may contain error handling if the processor accesses any of the addresses in the address range defined in the breakpoint registers.

For more information, see “Breakpoint Registers (PSx, DMx, IOx, and EPx)” on page 6-11.

Restrictions

Please note the following restriction when setting breakpoints.

- If a breakpoint interrupt comes at a point when a program is coming out of an interrupt service routine of a prior breakpoint, then in some cases the breakpoint status does not reflect that the second breakpoint interrupt has occurred.
- If an instruction address breakpoint is placed just after a short loop, a spurious breakpoint is generated.

Silicon Revision ID

The ADSP-2136x processor contains an 8-bit revision ID (REVPID), or the device identification register. This register can be read by using the JTAG instruction EMUPID. The I/O address of REVPID is 0x30026.

JTAG Related Registers

Information in this section describes public (JTAG) registers. These include:

- An instruction register, described on on page 6-6
- The EEMUSTAT register, described on on page 6-13
- Breakpoint registers, described on on page 6-11
- The EEMUIN register, described on on page 6-14
- The EEMUOUT register, described on on page 6-14
- The EMUCLK and EMUCLK2 registers, described on on page 6-14

Instruction Register

The instruction register shifts an instruction into the processor. This instruction selects the performed test and/or the access of the test data register. The instruction register is 5 bits long with no parity bit. A value of 11111 binary is loaded (LSB nearest TDO) into the instruction register whenever the TAP reset state is entered.

The new JTAG instruction set, shown in Table 6-2, lists the binary code for each instruction. Bit 0 is nearest TDO and bit 4 is nearest TDI. No data registers are placed into test modes by any of the public instructions. The instructions affect the processor as defined in the 1149.1 specification. The optional instructions RUNBIST, IDCODE, and USERCODE are not supported by the ADSP-2136x processor.

Table 6-2. JTAG Instruction Register Codes

43210	Register	Instruction	Inmode	Outmode
11111	Bypass	BYPASS	0	0
00000	Boundary	EXTEST	0	1
10000	Boundary	SAMPLE	0	0
11000	Boundary	INTEST	1	1
11100	BRKSTAT	EMULATION	0	0
01001	EEMUIN	EMULATION	0	0
01011	EEMUOUT	EMULATION	0	0
11101	EMUPID	REV-id register	0	0

The entry under "Register" is the serial scan path, either boundary or bypass, enabled by the instruction. Figure 6-1 shows these register paths. The 1-bit bypass register is fully defined in the 1149.1 specification.

No special values need to be written into any register prior to the selection of any instruction. As Table 6-2 shows, certain instructions are reserved for emulator use. For more information, see Figure 6-1.

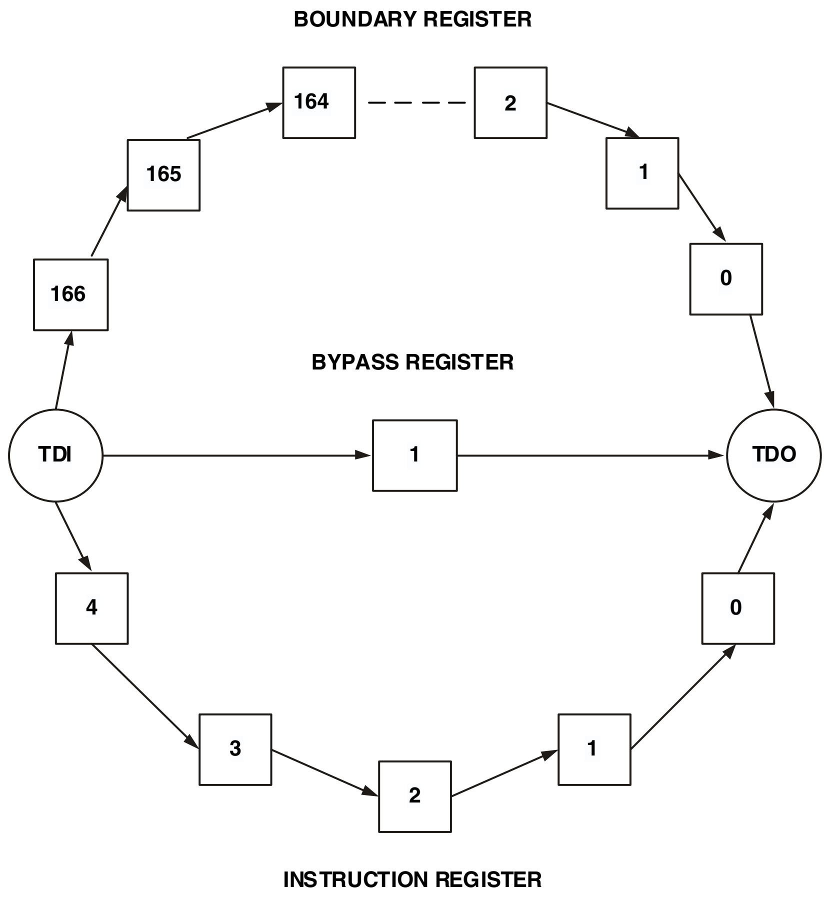


Figure 6-1. Serial Scan Path

Other registers, reserved for use by Analog Devices, exist. However, this group of registers should not be accessed as they can cause damage to the part.

Emulation Control Register (EMUCTL)

The `EMUCTL` serial shift register is located in the system unit. The `EMUCTL` register is 40 bits wide and is accessed by the emulator through the TAP. The `EMUCTL` register controls all of the ADSP-2136x processor emulation functionality. Table 6-3 lists the `EMUCTL` register's bits and describes their function.

Table 6-3. Emulation Control Register (EMUCTL) Descriptions

Bit	Name	Description
0	EMUENA	**Emulator Function Enable.** Enables processor emulation functions. 0 = ignore breakpoints and emulator interrupts 1 = respond to breakpoints and emulator interrupts)
1	EIRQENA	**Emulator Interrupt Enable.** Enables the emulation logic to create external emulator interrupts. (0 = disable, 1 = enable)
2	BKSTOP	**Enable Autostop on Breakpoint.** Enables the processor to generate an external emulator interrupt when any breakpoint event occurs. 0 = disable, 1 = enable.
3	SS	**Enable Single Step Mode.** Enables single-step operation. (0 = disable, 1 = enable)
4	SYSRST	**Software Reset.** Resets the processor in the same manner as the software reset bit in the SYSCTL register. The SYSRST bit must be cleared by the emulator. 0 = normal operation, 1 = reset.
5	ENBRKOUT	**Enable the BRKOUT pin.** Enables the $\overline{\text{BRKOUT}}$ pin operation. (0 = $\overline{\text{BRKOUT}}$ pin at high impedance state 1 = $\overline{\text{BRKOUT}}$ pin enabled.
6	IOSTOP	**Stop IOP DMAs in EMU Space.** Disables all DMA requests when the processor is in emulation space. Data that is currently in the EP, LINK, or SPORT DMA buffers is held there unless the internal DMA request was already granted. IOSTOP causes incoming data to be held off and outgoing data to cease. Because SPORT receive data cannot be held off, it is lost and the overrun bit is set. The direct write buffer (internal memory write) and the EP pad buffer are allowed to flush any remaining data to internal memory. 0 = I/O continues, 1 = I/O stops.

Table 6-3. Emulation Control Register (EMUCTL) Descriptions (Cont'd)

Bit	Name	Description
7	EPSTOP	**Stop I/O Processor EP operation in emulation space.** Disables all EP requests when the processor is in emulation space. After an emulation interrupt is acknowledged, EPSTOP deasserts ACK (deasserts REDY if host access) to prevent further data from being accepted if the EP is accessed. The emulator may clear this bit—allowing I/O to continue and the bus to clear—so that the emulator may use the EP (through BR and bus lock). Note that the EP bus clears only if accesses are direct writes or IOP register writes, because all other IOP functions are halted. The EP bus does not clear if accesses to any of the DMA buffers are extended due to a buffer full or empty condition. 0 = EP I/O continues, 1 = EP I/O stops.
8	NEGPA1[1]	**Negate program memory data address breakpoint.** Enable breakpoint events if the address is greater than the end register value OR less than the start register value. This function is useful to detect index range violations in user code. (0 = disable breakpoint, 1 = enable breakpoint)
9	NEGDA1	**Negate data memory address breakpoint #1** See NEGPA1 bit description.
10	NEGDA2	**Negate data memory address breakpoint #2.** See NEGPA1 bit description.
11	NEGIA1	**Negate instruction address breakpoint #1.** See NEGPA1 bit description.
12	NEGIA2	**Negate instruction address breakpoint #2.** See NEGPA1 bit description.
13	NEGIA3	**Negate instruction address breakpoint #3.** See NEGPA1 bit description.
14	NEGIA4	**Negate instruction address breakpoint #4.** See NEGPA1 bit description.
15	NEGIO1	**Negate I/O address breakpoint.** See NEGPA1 bit description.
16	NEGEP1	**Negate EP address breakpoint.** See NEGPA1 bit description.
17	ENBPA	**Enable program memory data address breakpoints.** Enable each breakpoint group. Note that when the ANDBKP bit is set, breakpoint types not involved in the generation of the effective breakpoint must be disabled. 0 = disable breakpoints, 1 = enable breakpoints.
18	ENBDA	**Enable data memory address breakpoints.** See ENBPA bit description.
19	ENBIA	**Enable instruction address breakpoints.** See ENBPA bit description.
20	Reserved	

Table 6-3. Emulation Control Register (EMUCTL) Descriptions (Cont'd)

Bit	Name	Description
21	ENBEP	**Enable external port address breakpoint** see ENBPA bit description.
22–23	PA1MODE	**PA1 breakpoint triggering mode.** Trigger on the following conditions: 00 = Breakpoint is disabled 01 = WRITE accesses only 10 = READ accesses only 11 = any access
24–25	DA1MODE	**DA1 breakpoint triggering mode**. See PA1MODE bit description.
26–27	DA2MODE	**DA2 breakpoint triggering mode.** See PA1MODE bit description.
28–29	IO1MODE	**IO1 breakpoint triggering mode.** See PA1MODE bit description.
30–31	EP1MODE	**EP1 breakpoint triggering mode.** See PA1MODE bit description.
32	ANDBKP	**AND composite breakpoints.** Enables ANDing of each breakpoint type to generate an effective breakpoint from the composite breakpoint signals. (0=OR breakpoint types, 1=AND breakpoint types)
33	Reserved	
34	NOBOOT	**No power-up boot on reset.** Forces the processor into the No boot mode. In this mode, the processor does not boot load, but begins fetching instructions from 0x0090 0004 in internal memory. (0 = disable, 1 = force No boot mode)
35	TMODE	**Test mode enable.** The TMODE bit is for Analog Devices' usage only. Do NOT set this bit. (0 = normal operation)
36	BHO	**Buffer Hang Override.** The BHO control bit overrides the Buffer Hang Disable (BHD) bit, disabling BHD's control over core access of data buffer behavior. (0 = normal BHD operation, 1 = override BHD operation)
37	MTST	**Memory Test Enable.** Enables scanning of data for to the latches used for memory test. (0 = normal operation, 1 = enable memory test)
38	ENBIOX	Enable IOX address breakpoint
39	ENBIOY	Enable IOY address breakpoint

1 Instruction address and program memory breakpoint negates have an effect latency of 4 core clock cycles.

Breakpoint Control Register (BRKCTL)

This BRKCTL register controls how breakpoints are used (if the UMODE bit is set). This user-accessible register is located at address 0x30025.

The BRKCTL register is a 32-bit memory-mapped I/O register. The core can write into this register and the bit information of this register is shown in the processor specific *ADSP-2136x SHARC Processor Hardware Reference*. The bits related to the breakpoint register are the same as in the EMUCTL register.

Breakpoint Registers (PSx, DMx, IOx, and EPx)

The PSx, DMx, IOx, and EPx (breakpoint) registers are located in the I/O processor register set. The emulation breakpoint registers are user-accessible if the UMODE bit is set in the BRKCTL register. Otherwise they can be written only when the ADSP-2136x processor is in emulation space or test mode. The breakpoint registers vary in size according to the address type: instruction (24-bit address), data (32-bit address), or I/O data (19-bit address).

The ADSP-2136x processor contains nine sets of emulation Breakpoint registers. Each set consists of a start and end register which describe an address range, with the start register setting the lower end of the address range. Each breakpoint set monitors a particular address bus. When a valid address is in the address range, then a breakpoint signal is generated. The address range includes the start and end addresses.

The nine breakpoint sets are grouped into four types—instruction (IA), DM data (DA), PM data (PA), and I/O data (I/O). The individual breakpoint signals in each type are ORed together to create five composite breakpoint signals.

These composite signals can be optionally ANDed or ORed together to create the effective breakpoint event signal used to generate an emulator interrupt. The ANDBKP bit in the EMUCTL register selects the function used.

Each breakpoint type has an enable bit in the EMUCTL register. When set, these bits add the specified breakpoint type into the generation of the effective breakpoint signal. If cleared, the specified breakpoint type is not used in the generation of the effective breakpoint signal. This allows the user to trigger the effective breakpoint from a subset of the breakpoint types.

To provide further flexibility, each individual breakpoint can be programmed to trigger if the address is in range AND one of these conditions is met: READ access, WRITE access, ANY access, or NO access. The control bits for this feature are also located in EMUCTL register. For more information, see the PA1MODES bit description in Table 6-4.

The address ranges of the emulation breakpoint registers are negated by setting the appropriate negation bits in the EMUCTL register. For more information, see the NEGPA1 bit description in Table 6-3 on page 6-8. Each breakpoint can be disabled by setting the start address larger than the end address.

Four of the breakpoints monitor the instruction address, two monitor the data memory address, one monitors the program memory data address, and two monitor the I/O address bus.

The instruction address breakpoints monitor the address of the instruction being executed, not the address of the instruction being fetched. If the current execution is aborted, the breakpoint signal does not occur even if the address is in range. Data address breakpoints (DA and PA only) are also ignored during aborted instructions. The nine breakpoint sets appear in Table 6-4.

Table 6-4. PSx, DMx, IOx, and EPx (Breakpoint) Registers

Register	Function	Group[1]
PSA1S	Instruction Address Start #1	IA
PSA1E	Instruction Address End #1	IA

Table 6-4. PSx, DMx, IOx, and EPx (Breakpoint) Registers (Cont'd)

Register	Function	Group[1]
PSA2S	Instruction Address Start #2	IA
PSA2E	Instruction Address End #2	IA
PSA3S	Instruction Address Start #3	IA
PSA3E	Instruction Address End #3	IA
PSA4S	Instruction Address Start #4	IA
PSA4E	Instruction Address End #4	IA
DMA1S	Data Address Start #1	DA
DMA1E	Data Address End #1	DA
DMA2S	Data Address Start #2	DA
DMA2E	Data Address End #2	DA
PMDAS	Program Data Address Start	PA
PMDAE	Program Data Address End	PA
IOAS	I/O Address Start	I/O
IOAE	I/O Address End	I/O
EPAS	External Port Address Start	EP

1 Group IA = 24-bit addresses, Groups DA and PA = 32-bit addresses, Group I/O = 19-bit addresses.

Enhanced Emulation Status Register (EEMUSTAT)

The `EEMUSTAT` register acts as the breakpoint status register for the ADSP-2136x processor. This register is a memory-mapped IOP register. The processor core can access this register. For I/O breakpoints, this register has two status bits, one each for the two I/O buses (`IOX` and `IOY`).

When a breakpoint is hit, a user interrupt is generated. The breakpoint status can be checked by looking at the `EEMUSTAT` register. When the core returns from interrupt, the breakpoint status bits are cleared.

EEMUIN Register

The EEMUIN register is a one-deep, 32-bit memory-mapped I/O buffer that is readable by the core. This buffer is used by the background telemetry channel to allow the emulator to pass data to the processor without interrupting the core. When this buffer is full, a low priority emulator interrupt is generated. This register's address is 0x30020.

EEMUOUT Register

The EEMUOUT register is a four-deep memory, 32-bit memory-mapped I/O buffer that is writable by the core. Its address is 0x30022.

Emulation Clock Counter Registers (EMUCLK, EMUCLK2)

The EMUCLK (clock counter) and EMUCLK2 (clock counter scaling) registers are located in the universal (*Ureg*) register set. EMUCLK and EMUCLK2 registers are user accessible and can be written only when the processor is in emulation space. These registers are read-only from normal-space and can be written only when the ADSP-2136x processor is in emulation space. The emulation clock counter consists of a 32-bit count register (EMUCLK) and a 32-bit scaling register (EMUCLK2). The EMUCLK register counts clock cycles while the user has control of the processor and stops counting when the emulator gains control. These registers let you gauge the amount of time spent executing a particular section of code. The EMUCLK2 register extends the time EMUCLK can count by incrementing each time the EMUCLK value rolls over to zero. The combined emulation clock counter can count accurately for thousands of hours.

When the emulator is connected to the processor and the processor is single stepping, extra cycles are used by the emulator and this can make it seem as though the instructions are taking more cycles then they should.

You can see the actual cycle time of the processor (without the emulator) by polling the `EMUCLK` and `EMUCLK2` registers. The processor cycle count can be seen while the core is in user space.

Boundary Register

The boundary-scan register is selected by the `EXTEST`, `INTEST`, and `SAMPLE` instructions. These instructions allow the pins of the processor to be controlled and sampled for board-level testing. For the most recent BSDL files, please visit the Analog Devices Web site.

EMUN Register

The `EMUN` (Nth event counter) register is located in the I/O processor register set. The `EMUN` register can only be written to if the `BRKCTL` bit is set. The Nth event counter allows an emulation breakpoint to occur on the Nth occurrence of the breakpoint event. This is accomplished by writing the desired Nth value to the `EMUN` register in *`UREG`* space. The counter decrements on each occurrence of the breakpoint event, asserting the interrupt when the counter is equal to zero and the hardware breakpoint event occurs. For more information, see the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* or the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

EMUIDLE Instruction

The `EMUIDLE` instruction places the ADSP-2136x processor in the idle state and triggers an emulator interrupt. This operation uses the `EMUIDLE` instruction as a software breakpoint. When the `EMUIDLE` instruction is executed, the emulation clock counter immediately halts. For more information, see the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* or the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

Operating System Process ID Register (OSPID)

The `OSPID` register is a 32-bit memory-mapped I/O register that is sampled into the shift register, `EMUOSPID`, whenever the program counter is sampled into the `EMUPC` register. The `EMUOSPID` and `EMUPC` registers form a single scan chain.

The `OSPID` feature is controlled using the (`OSPIDEN`, bit 1) in the enhanced emulation control/status register. To enable this feature, set this bit = 1. If not enabled (= 0), the legacy feature is supported.

The operation is explained in the following steps.

1. The new feature is enabled by setting the `OSPIDEN` bit in the enhanced emulation control register. The enhanced emulation enable bit need not be set to enable this feature.
2. Whenever the TAP controller returns to the `RUNTEST` state, the contents of the program counter are sampled into the `EMUPC` register. The `OSPID` register is also loaded into the `EMUOSPID` register.
3. Both the `EMUPC` and `EMUOSPID` registers can be selected by the same JTAG instruction (instruction for the `EMUPC` register), since they form a single scan chain.
4. The TAP controller sends the `CAPTURE` signal to both the register and status bits of the `EMUOSPID` and `EMUPC` registers into shift registers.
5. The TAP enters to the `SHIFT` state and shifts out 56-bit data. In this case, the first 32 bits indicate program id, and the last 24 bits provide the address of instruction executed in that program id.

Private Instructions

Table 6-2 lists the private instructions that are reserved for emulation and memory test. The ADSP-21364 EZ-KIT Lite emulator uses the TAP and boundary scan as a way to access the processor in the target system. The EZ-KIT emulator requires a target board connector for access to the TAP.

References

- IEEE Standard 1149.1-1990. Standard Test Access Port and Boundary-Scan Architecture.

  To order a copy, contact IEEE at 1-800-678-IEEE.

- Maunder, C.M. and R. Tulloss. Test Access Ports and Boundary Scan Architectures.

  IEEE Computer Society Press, 1991.

- Parker, Kenneth. The Boundary Scan Handbook.

  Kluwer Academic Press, 1992.

- Bleeker, Harry, P. van den Eijnden, and F. de Jong. Boundary-Scan Test—A Practical Approach.

  Kluwer Academic Press, 1993.

- Hewlett-Packard Co. HP Boundary-Scan Tutorial and BSDL Reference Guide.

  (HP part# E1017-90001) 1992.

7 TIMER

In addition to the internal core timer, the ADSP-2136x processor contains three identical 32-bit timers that can be used to interface with external devices. Each timer can be individually configured in any of three modes:

- "Pulse Width Modulation Mode (PWM_OUT)" on page 7-9
- "Pulse Width Count and Capture Mode (WDTH_CAP)" on page 7-12
- "External Event Watchdog Mode (EXT_CLK)" on page 7-14

Timer Architecture

Each timer has one dedicated bidirectional chip signal, TIMERx. The three timer signals are connected to the 14 digital peripheral interface (DPI) pins through the signal routing unit 2 (SRU2). The timer signal functions as an output signal in PWM_OUT mode and as an input signal in WDTH_CAP and EXT_CLK modes. To provide these functions, each timer has four, 32-bit registers. The registers for each timer are:

- Timer x control (TMxCTL) register
- Timer x word count (TMxCNT) register
- Timer x word period (TMxPRD) register
- Timer x word pulse width (TMxW) register

The timers also share a common status and control register—the timer global status and control (TMSTAT) register.

For information on the timer registers, see "Timer Registers" on page B-35.

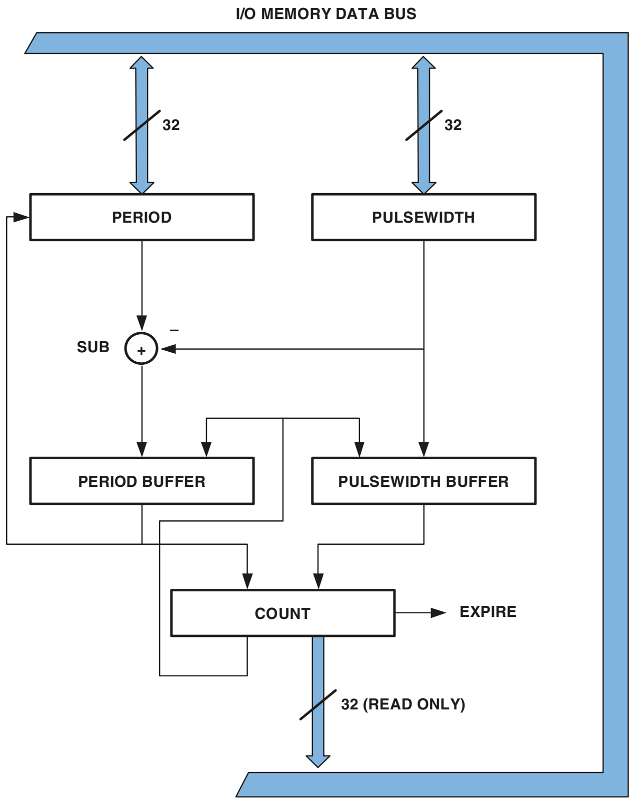


Figure 7-1. Timer Block Diagram

When clocked internally, the clock source is the ADSP-2136x processor's peripheral clock (PCLK). The timer produces a waveform with a period equal to 2 x TMxPRD and a width equal to 2 x TMxW. The period and width are set through the TMxPRD[30:0] and the TMxW[30:0] bits. Bit 31 is ignored for both. Assuming HCLK=166 MHz:

maximum period = 2 x (2^{31} – 1) x 6 ns = 40 seconds.

Timer and Sequencing

The sequencer is attached to the programmable interval timer. Bits in the MODE2, TCOUNT, and TPERIOD registers control timer operations as described below.

- **Timer enable** MODE2 Bit 5 (TIMEN). This bit directs the processor to enable (if 1) or disable (if 0) the timer.
- **Timer count** (TCOUNT). This register contains the decrementing timer count value, counting down the cycles between timer interrupts.
- **Timer period** (TPERIOD). This register contains the timer period, indicating the number of cycles between timer interrupts.

Table B-3 on page B-11 lists all of the bits in the MODE2 register.

The TCOUNT register contains the timer counter. The timer decrements the TCOUNT register during each clock cycle. When the TCOUNT value reaches zero, the timer generates an interrupt and asserts the FLAGx pin. This scenario applies only when FLAG3 is configured as TIMEXP. On the clock cycle after TCOUNT reaches zero, the timer automatically reloads TCOUNT from the TPERIOD register.

The TPERIOD value specifies the frequency of timer interrupts. The number of cycles between interrupts is TPERIOD + 1. The maximum value of TPERIOD is $2^{32} - 1$. This value is loaded into TCOUNT after it decrements to zero.

To start and stop the timer, programs use the MODE2 register's TIMEN bit. With the timer disabled (TIMEN=0), the program loads TCOUNT with an initial count value and loads TPERIOD with the number of cycles for the desired interval. Then, the program enables the timer (TIMEN=1) to begin the count.

When a program enables the timer, the timer starts decrementing the TCOUNT register at the end of the next two clock cycles after the current cycle. If the timer is subsequently disabled, the timer stops decrementing TCOUNT one clock cycle after the current cycle as shown in Figure 7-2.

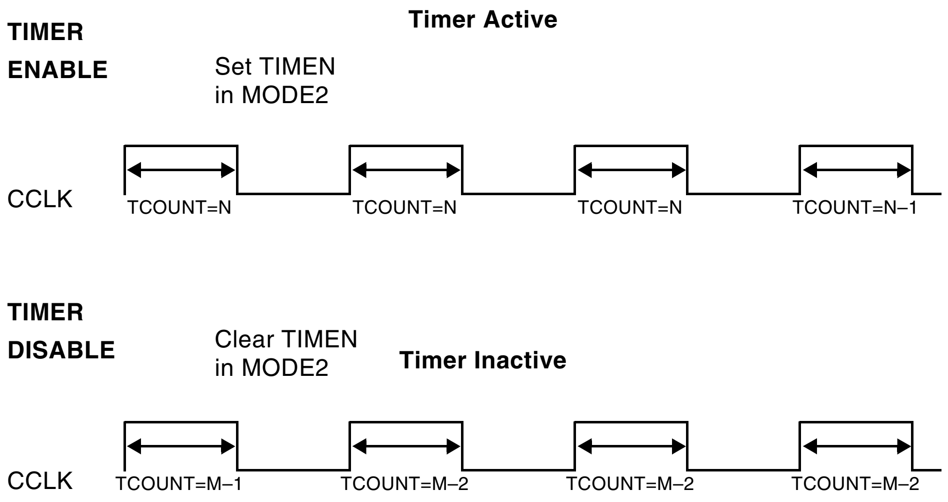


Figure 7-2. Timer Enable and Disable

The timer expired event (TCOUNT decrements to zero) generates two interrupts, TMZHI and TMZLI. For information on latching and masking these interrupts to select timer expired priority, see "Latching Interrupts" on page 3-76.

As with other interrupts, the sequencer needs a minimum of five cycles to fetch and decode the first instruction of the timer expired service routine before executing the routine.

See Table 3-39 on page 3-71, Table 3-40 on page 3-72, and Table 3-41 on page 3-72 for pipelined execution cycles for interrupt processing.

Programs can read and write the TPERIOD and TCOUNT registers by using universal register transfers. Reading the registers does not effect the timer. Note that an explicit write to TCOUNT takes priority over the sequencer's loading TCOUNT from TPERIOD and the timer's decrementing of TCOUNT. Also note that TCOUNT and TPERIOD are not initialized at reset. Programs should initialize these registers before enabling the timer.

Timer Status and Control

The timer global status and control (TMSTAT) register indicates the status of all three timers using a single read. The TMSTAT register also contains timer enable bits. Within TMSTAT, each timer has a pair of sticky status bits, that require a write one-to-set (TIMxEN) or write one-to-clear (TIMxDIS) to enable and disable the timer respectively.

Writing a one to both bits of a pair disables that timer.

Each timer also has an overflow error detection bit, TIMxOVF. When an overflow error occurs, this bit is set in the TMSTAT register. A program must write one-to-clear this bit.

See Table 7-1 for more information about bits in the TMSTAT register.

Table 7-1. Timer Global Status and Control (TMSTAT) Register Bits

Bit	Name	Description
0	TIM0IRQ Timer 0 Interrupt Latch	Write one-to-clear (also an output)[1]
1	TIM1IRQ Timer 1 Interrupt Latch	Write one-to-clear (also an output)1
2	TIM2IRQ Timer 2 Interrupt Latch	Write one-to-clear (also an output)1
3	Reserved	
4	TIM0OVF Timer 0 Overflow/Error	Write one-to-clear (also an output)
5	TIM1OVF Timer 1 Overflow/Error	Write one-to-clear (also an output)
6	TIM2OVF Timer 2 Overflow/Error	Write one-to-clear (also an output)
7	Reserved	
8	TIM0EN Timer 0 Enable	Write one-to-enable Timer 0
9	TIM0DIS Timer 0 Disable	Write one-to-disable Timer 0
10	TIM1EN Timer 1 Enable	Write one-to-enable Timer 1
11	TIM1DIS Timer 1 Disable	Write one-to-disable Timer 1
12	TIM2EN Timer 2 Enable	Write one-to-enable Timer 2
13	TIM2DIS Timer 2 Disable	Write one-to-disable Timer 2
31–14	Reserved	

1 This bit is set to one when an interrupt generating event occurs. When the program writes a one to this bit position, it clears the source event which causes this bit to clear. A subsequent read of this bit returns a zero.

After the timer has been enabled, its `TIMxEN` bit is set (= 1). The timer then starts counting three peripheral clock cycles (`PCLK`) after the `TIMxEN` bit is set. Setting (writing one to) the timer's `TIMxDIS` bit stops the timer without waiting for another event.

Timer Interrupts

Each timer generates a unique interrupt request signal. A common register latches these interrupts so that a program can determine the interrupt source without reference to the timer's interrupt signal. The `TMSTAT` register contains an interrupt latch bit (`timxirq`) and an overflow/error indicator bit (`TIMxOVF`) for each timer.

The three timer interrupts are connected as follows:

- `TIM0IRQ` to `GPTMR0I`, bit 13 in the `IRPTL` register
- `TIM1IRQ` to `GPTMR1I`, bit 4 in the `LIRPTL` register
- `TIM2IRQ` to `GPTMR2I`, bit 8 in the `LIRPTL` register

These sticky bits are set by the timer hardware and may be watched by software. They need to be cleared in the `TMSTAT` register by software explicitly. To clear, write a one to the corresponding bit in the `TMSTAT` register.

ⓘ Interrupt and overflow bits may be cleared simultaneously with timer enable or disable.

To enable a timer's interrupt, set the `IRQEN` bit in the timer's configuration (`TMxCTL`) register and unmask the timer's interrupt by setting the corresponding bit of the `IMASK` register. With the `IRQEN` bit cleared, the timer does not set its interrupt latch (`TIMxIRQ`) bits. To poll the `TIMxIRQ` bits without generating a timer interrupt, programs can set the `IRQEN` bit while leaving the timer's interrupt masked.

With interrupts enabled, ensure that the interrupt service routine (ISR) clears the `TIMxIRQ` latch before the `RTI` instruction to assure that the interrupt is not serviced erroneously. In external clock (`EXT_CLK`) mode, the latch should be reset at the very beginning of the interrupt routine so as not to miss any timer event.

Enabling a Timer

To enable an individual timer, set the timer's `TIMxEN` bit in the `TMSTAT` register. To disable an individual timer, set the timer's `TIMxDIS` bit in the `TMSTAT` register. To enable all three timers in parallel, set all the `TIMxEN` bits in the `TMSTAT` register.

Before enabling a timer, always program the corresponding timer's configuration (`TMxCTL`) register. This register defines the timer's operating mode, the polarity of the `TIMERx` signal, and the timer's interrupt behavior. Do not alter the operating mode while the timer is running. For more information on the `TMxCTL` register, see "Timer Configuration Registers (TMxCTL)" on page B-35.

The timer enable and disable timing for `PWM_OUT` appears in Figure 7-3.

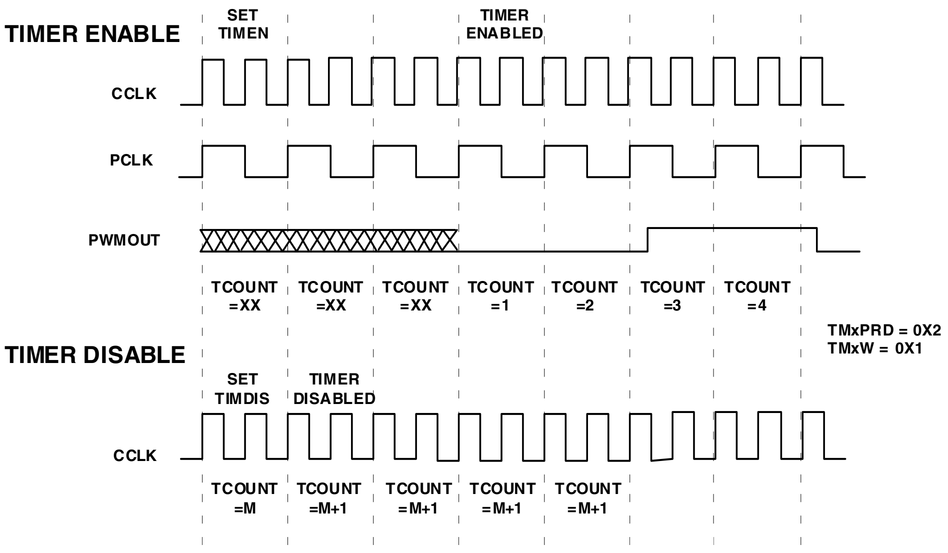


Figure 7-3. Timer PWM Enable and Disable Timing

When the timer is enabled, the count register is loaded according to the operation mode specified in the TMxCTL register. When the timer is disabled, the counter registers retain their state; when the timer is re-enabled, the counter is reinitialized based on the operating mode. The software should never write the counter value directly.

Any of the timers can be used to implement a watchdog functionality that can be controlled by either an internal or an external clock source.

For software to service the watchdog, the program must reset the timer value by disabling and then re-enabling the timer. Servicing the watchdog periodically prevents the count register from reaching the period value and prevents the timer interrupt from being generated. When the timer reaches the period value and generates the interrupt, reset the processor within the corresponding watchdog's ISR.

Pulse Width Modulation Mode (PWM_OUT)

In PWM_OUT mode, the timer supports on-the-fly updates of period and width values of the PWM waveform. The period and width values can be updated once every PWM waveform cycle, either within or across PWM cycle boundaries.

To enable PWM_OUT mode, set the TIMODE1-0 bits to 01 in the timer's Configuration (TMxCTL) register. This configures the timer's TIMERx signal as an output with its polarity determined by PULSE as follows:

- If PULSE is set (= 1), an active high width pulse waveform is generated at the TIMERx signal.
- If PULSE is cleared (= 0), an active low width pulse waveform is generated at the TIMERx signal.

The timer is actively driven as long as the TIMODE field remains 01.

Figure 7-4 shows a flow diagram for `PWM_OUT` mode. When the timer becomes enabled, the timer checks the period and width values for plausibility (independent of the value set with the `PRDCNT` bit) and does *not* start to count when any of the following conditions are true:

- Width is equal to zero
- Period value is lower than width value
- Width is equal to period

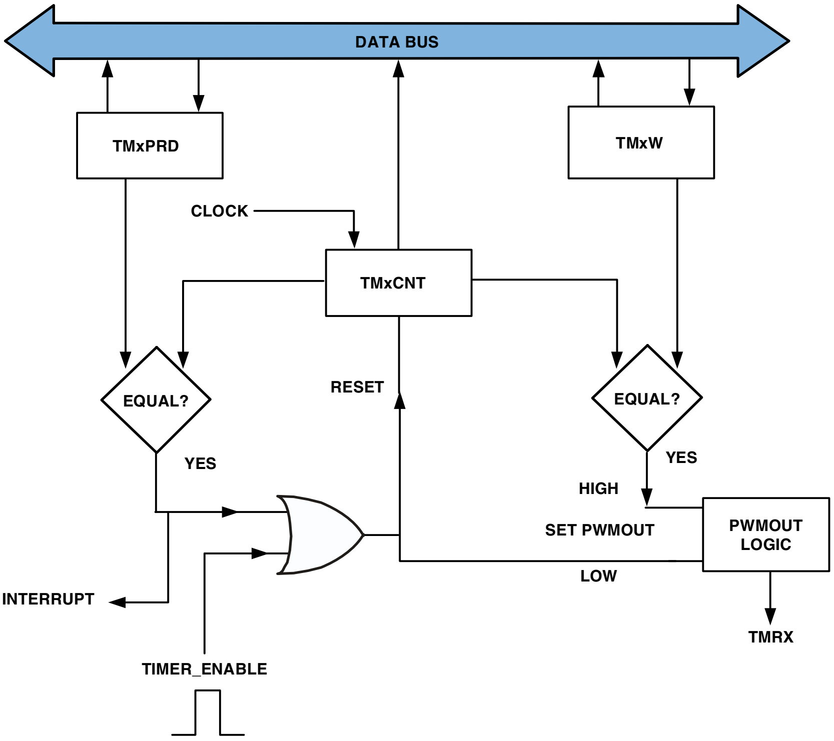


Figure 7-4. Timer Flow Diagram – PWM_OUT Mode

On invalid conditions, the timer sets both the TIMxOVF and the TIMIRQx bits and the Count register is not altered. Note that after reset, the timer registers are all zero.

As mentioned earlier, 2 x TMxPRD is the period of the PWM waveform and 2 x TMxW is the width. If the period and width values are valid after the timer is enabled, the Count register is loaded with the value resulting from 0xFFFF FFFF – width. The timer counts upward to 0xFFFF FFFF. Instead of incrementing to 0xFFFF FFFF, the timer then reloads the counter with the value derived from 0xFFFF FFFF – (period – width) and repeats.

PWM Waveform Generation

If the PRDCNT bit is set, the internally-clocked timer generates rectangular signals with well-defined period and duty cycles. This mode also generates periodic interrupts for real-time processing.

The 32-bit period (TMxPRD) and width (TMxW) registers are programmed with the values of the timer count period and pulse width modulated output pulse width.

When the timer is enabled in this mode, the TIMERx signal is pulled to a deasserted state each time the pulse width expires, and the signal is asserted again when the period expires (or when the timer is started).

To control the assertion sense of the TIMERx signal, the PULSE bit in the corresponding TMxCTL register is either cleared (causes a low assertion level) or set (causes a high assertion level).

When enabled, a timer interrupt is generated at the end of each period. An ISR must clear the interrupt latch bit TIMxIRQ and might alter period and/or width values. In pulse width modulation applications, the software needs to update the period and pulse width values while the timer is running.

When a program updates the timer configuration, the TMxW register must always be written to last, even if it is necessary to update only one of the registers. When the TMxW value is not subject to change, the ISR reads the current value of the TMxW register and rewrite it again. On the next counter reload, all of the timer control registers are read by the timer.

To generate the maximum frequency on the TIMERx output signal, set the period value to two and the pulse width to one. This makes the TIMERx signal toggle every four CCLK or 2 PCLK (peripheral clock period = 2 x CCLK) clock cycles as shown in Figure 7-3.

Single-Pulse Generation

If the PRDCNT bit is cleared, the PWM_OUT mode generates a single pulse on the TIMERx signal. This mode can also be used to implement a well defined software delay that is often required by state machines. The pulse width (= 2 x TMxW) is defined by the width register and the period register is not used.

At the end of the pulse, the interrupt latch bit (TIMxIRQ) is set and the timer is stopped automatically. If the PULSE bit is set, an active high pulse is generated on the TIMERx signal. If the PULSE bit is not set, the pulse is active low.

Pulse Width Count and Capture Mode (WDTH_CAP)

To enable WDTH_CAP mode, set the TIMODE1-0 bits in the TMxCTL register to 10. This configures the TIMERx signal as an input signal with its polarity determined by PULSE. If PULSE is set (= 1), an active high width pulse waveform is measured at the TIMERx signal. If PULSE is cleared (= 0), an active low width pulse waveform is measured at the TIMERx signal. The internally-clocked timer is used to determine the period and pulse width of externally-applied rectangular waveforms. The period and width registers are read-only in WDTH_CAP mode. The period and pulse width measurements are with respect to a clock frequency of CCLK/4 or PCLK/2.

Figure 7-5 shows a flow diagram for WDTH_CAP mode. In this mode, the timer resets words of the count in the TMxCNT register value to 0x0000 0000 and does not start counting until it detects the leading edge on the TIMERx signal.

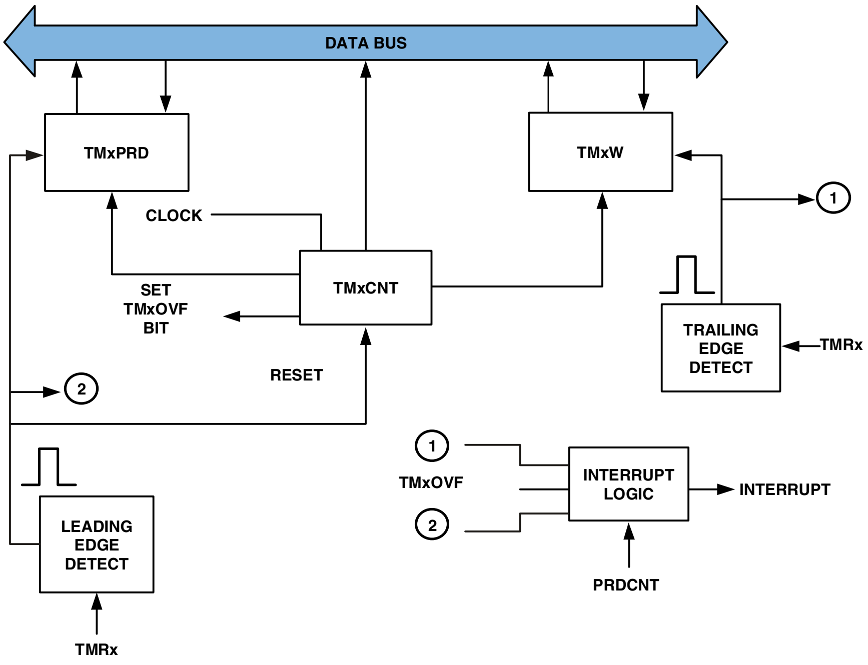


Figure 7-5. Timer Flow Diagram – WDTH_CAP Mode

When the timer detects a first leading edge, it starts incrementing. When it detects the trailing edge of a waveform, the timer captures the current value of the count register (= TMxCNT/2) and transfers it into the TMxW width registers. At the next leading edge, the timer transfers the current value of the count register (= TMxCNT/2) into the TMxPRD period register. The count registers are reset to 0x0000 0000 again, and the timer continues counting until it is either disabled or the count value reaches 0xFFFF FFFF.

In this mode, software can measure both the pulse width and the pulse period of a waveform. To control the definition of the leading edge and trailing edge of the TIMERx signal, the PULSE bit in the TMxCTL register is set or cleared. If the PULSE bit is cleared, the measurement is initiated by a falling edge, the count register is captured to the Width register on the rising edge, and the period register is captured on the next falling edge.

The PRDCNT bit in the TMxCTL register controls whether an enabled interrupt is generated when the pulse width or pulse period is captured. If the PRDCNT bit is set, the interrupt latch bit (TIMxIRQ) gets set when the pulse period value is captured. If the PRDCNT bit is cleared, the TIMxIRQ bit gets set when the pulse width value is captured.

If the PRDCNT bit is cleared, the first period value has not yet been measured when the first interrupt is generated. Therefore, the period value is not valid. If the interrupt service routine reads the period value anyway, the timer returns a period value of zero. When the period expires, the period value is loaded in the TMxPRD register.

A timer interrupt (if enabled) is also generated if the count register reaches a value of 0xFFFF FFFF. At that point, the timer is disabled automatically, and the TIMxOVF Status bit is set, indicating a count overflow. The TIMxIRQ and TIMxOVF bits are sticky bits, and software must explicitly clear them.

The first width value captured in WDTH_CAP mode is erroneous due to synchronizer latency. To avoid this error, software must issue two NOP instructions between setting WDTH_CAP mode and setting TIMxEN.

External Event Watchdog Mode (EXT_CLK)

To enable EXT_CLK mode, set the TIMODE1-0 bits in the TMxCTL register to 11 in the TMxCTL register. This configures the TIMERx signal as an input. The PULSE bit determines the TIMERx signal polarity. The timer works as a counter clocked by any external source, which can also be asynchronous to the processor clock. Therefore, in EXT_CLK mode, the TMxCNT register should not be read when the counter is running.

The operation of the EXT_CLK mode is as follows:

1. Program the TMxPRD period register with the value of the maximum timer external count.

2. Set the TIMxEN bits. This loads the period value in the count register and starts the countdown.

3. When the period expires, an interrupt, (TIMxIRQ) occurs.

After the timer is enabled, it waits for the first rising edge on the TIMERx signal. The PULSE bit defines the rising edge and trailing edge. The rising edge forces the count register to be loaded by the value (0xFFFF FFFF – TMxPRD). Every subsequent rising edge increments the count register. After reaching the count value 0xFFFF FFFE, the TIMxIRQ bit is set and an interrupt is generated. The next rising edge reloads the count register with (0xFFFF FFFF – TMxPRD) again.

The configuration bit, PRDCNT, has no effect in this mode. Also, TIMxOVF is never set and the width register is unused.

Timer Programming Examples

This section provides two programming examples written for the ADSP-2136x processors.

The first listing, Listing 7-1, sets up timer 0 in external watchdog mode, using DAI pin 1 as its input. The timer generates an interrupt when it senses the number of edges are equal to the timer period setting. The second listing, Listing 7-2, uses both timer 0 and timer 1. Timer 0 is set up in PWMOUT mode, using DAI pin 1 as its output. Timer 1 is set up in width capture mode, using Timer 0 as its input. The period and pulse width measured by timer 1 are identical to the settings of timer 0.

Listing 7-1. External Watchdog Mode Example

```
/* Register Definitions */
#define TMSTAT  (0x1400)   /* GP Timer 0 Status register  */
#define TM0CTL  (0x1401)   /* GP Timer 0 Control register */
#define TM0PRD  (0x1403)   /* GP Timer 0 Period register  */
#define TM0W    (0x1404)   /* GP Timer 0 Width register   */
#define SRU_EXT_MISCB    (0x2471)


/* SRU definitions */
#define DAI_PB01_O       0x00


/* Bit Positions */
#define TIMER0_I        0

/* Bit Definitions */
#define TIMODEEXT 0x00000003
#define PULSE     0x00000004
#define PRDCNT    0x00000008
#define IRQEN     0x00000010
#define TIM0EN    0x00000100


/* Main code section */
.global _main;
.section/pm seg_pmco;
_main:


/* Route Timer 0 Input to DAI Pin 1 via SRU */
```

```
r0 = (DAI_PB01_O<<TIMER0_I);
dm(SRU_EXT_MISCB)=r0;
ustat3 = TIMODEEXT|     /* External Watchdog Mode */
         PULSE|         /* Positive edge is active */
         IRQEN|         /* Enable Timer 0 interrupt */
         PRDCNT;        /* Count to end of period */
dm(TM0CTL) = ustat3;
R0 = 0xff;
dm(TM0PRD) = R0;        /* Timer 0 period = 255 */
/* An interrupt is generated when the Timer senses end of the
selected period, In this example Interrupts are disabled, so pro-
gram flow will not be affected */

R0 = TIM0EN;            /* Enable timer 0 */
dm(TMSTAT) = R0;


_main.end: jump (pc,0);  /* endless loop */
```

Listing 7-2. PWMOUT and Width Capture Mode Example

```
/* Register Definitions */
#define TMSTAT  (0x1400)   /* GP Timer 0 Status register  */
#define TM0CTL  (0x1401)   /* GP Timer 0 Control register */
#define TM0CNT  (0x1402)   /* GP Timer 0 Count register   */
#define TM0PRD  (0x1403)   /* GP Timer 0 Period register  */
#define TM0W    (0x1404)   /* GP Timer 0 Width register   */
#define TM1CTL  (0x1409)   /* GP Timer 1 Control register */
#define TM1CNT  (0x140A)   /* GP Timer 1 Count register   */
#define TM1PRD  (0x140B)   /* GP Timer 1 Period register  */
#define TM1W    (0x140C)   /* GP Timer 1 Width register   */
#define SRU_PIN0           (0x2460)
#define SRU_PBEN0          (0x2478)
#define SRU_EXT_MISCB      (0x2471)
```

```
/* Bit Definitions */
#define TIMODEPWM 0x00000001
#define TIMODEW   0x00000002
#define PULSE     0x00000004
#define PRDCNT    0x00000008
#define IRQEN     0x00000010
#define TIM0EN    0x00000100
#define TIM1EN    0x00000400
#define GPTMR1I   0x00000010


/* SRU Definitions */
#define TIMER0_Od       0x2C
#define TIMER0_Oe       0x14
#define PBEN_HIGH_Of    0x01
/* Bit positions */
#define TIMER1_I        5


/* Main code section */
.global _main;
.section/pm seg_pmco;
_main:
/* Set up and enable Timer 0 in PWM Out mode*/
/* Route Timer 0 Output to DAI Pin 1 via SRU */

r0 = TIMER0_Od;dm(SRU_PIN0) = r0;
/* Enable DAI pin 1 as an output */
r0 = PBEN_HIGH_Of;
dm(SRU_PBEN0) = r0;
ustat3 = TIMODEPWM|      /* PWM Out Mode */
         PULSE|          /* Positive edge is active */
         PRDCNT;         /* Count to end of period */
dm(TM0CTL) = ustat3;
```

```
R0 = 0xFF;
dm(TM0PRD) = R0;       /* Timer 0 period = 255 */
R1 = 0x3F;
dm(TM0W) = R1;         /* Timer 0 Pulse width = 15 */

R0 = TIM0EN;           /* enable timer 0 */
dm(TMSTAT) = R0;

/* --------------End of Timer 0 Setup-------------------- */

/* Set up and enable Timer 1 in Width Capture mode */
/* Use the output of Timer 0 as the input to Timer 1 */
/* Route Timer 0 Output to Timer 1 Input via SRU */
r0=(TIMER0_Oe<<TIMER1_I);
dm(SRU_EXT_MISCB)=r0;

ustat3 = TIMODEW|       /* Width Capture mode */
         PULSE|         /* Positive edge is active */
         IRQEN|         /* Enable Timer 1 Interrupt */
         PRDCNT;        /* Count to end of period */
dm(TM1CTL) = ustat3;

R0 = TIM1EN;            /* enable timer 1 */
dm(TMSTAT) = R0;

/* Poll the Timer 1 interrupt latch, the interrupt will latch
when the measured period and pulse width are ready to read */
bit tst LIRPTL GPTMR1I;
if not tf jump(pc,-1);
```

```
/* Read the measured values */
r0 = dm(TM1PRD);
r1 = dm(TM1W);
/* r0 and r1 will match the Timer 0 settings above */
_main.end: jump (pc,0);
```

8 INSTRUCTION SET

The compute and move instructions in the Group I set of instructions specify a compute operation in parallel with one or two data moves or an index register modify.

Group I Instructions

The instructions in this group contain a COMPUTE field that specifies a compute operation using the ALU, multiplier, or shifter. Because there are a large number of options available for computations, these operations are described separately in “Computations Reference” in Chapter 9, Computations Reference. Note that data moves between the MR registers and the register file are considered multiplier operations and are also covered in “Computations Reference” in Chapter 9, Computations Reference. Group I instructions include the following.

- “Type 1: Compute, Dreg«···»DM | Dreg«···»PM” on page 8-3

  Parallel data memory and program memory transfers with register file, optional compute operation

- “Type 2: Compute” on page 8-6

  Compute operation, optional condition

- “Type 3: Compute, ureg«···»DM | PM, register modify” on page 8-8

  Transfer between data or program memory and universal register, optional condition, optional compute operation

- “Type 4: Compute, dreg«···»DM | PM, data modify” on page 8-13

  PC-relative transfer between data or program memory and register file, optional condition, optional compute operation

- “Type 5: Compute, ureg«··· »ureg | Xdreg<->Ydreg” on page 8-18

  Transfer between two universal registers, optional condition, optional compute operation

- “Type 6: Immediate Shift, dreg«···»DM | PM” on page 8-22

  Immediate shift operation, optional condition, optional transfer between data or program memory and register file

- “Type 7: Compute, modify” on page 8-27

  Index register modify, optional condition, optional compute operation

Type 1: Compute, Dreg«··»DM | Dreg«··»PM

Parallel data memory and program memory transfers with register file, option compute operation

Syntax

```
compute   |, DM(Ia, Mb) = dreg1   |   |, PM(Ic, Md) = dreg2   | ;
          |, dreg1 = DM(Ia, Mb)   |   |, dreg2 = PM(Ic, Md)   |
```

Function (SISD)

In SISD mode, the Type 1 instruction provides parallel accesses to data and program memory from the register file. The specified I registers address data and program memory. The I values are post-modified and updated by the specified M registers. Pre-modify offset addressing is not supported. For more information on register restrictions, see "Data Address Generators" in Chapter 4, Data Address Generators.

Function (SIMD)

In SIMD mode, the Type 1 instruction provides the same parallel accesses to data and program memory from the register file as are available in SISD mode, but provides these operations simultaneously for the X and Y processing elements.

The X element uses the specified I registers to address data and program memory, and the Y element adds one to the specified I registers to address data and program memory. If the broadcast read bits—`BDCST1` (for `I1`) or `BDCST9` (for `I9`)—are set, the Y element uses the specified I register without adding one.

The I values are post-modified and updated by the specified M registers. Pre-modify offset addressing is not supported. For more information on register restrictions, see "Data Address Generators" in Chapter 4, Data Address Generators.

The X element uses the specified `Dreg` registers, and the Y element uses the complementary registers (`Cdreg`) that correspond to the `Dreg` registers. For a list of complementary registers, see Table 2-16 on page 2-47.

The following pseudo code compares the Type 1 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

```
compute   |, DM(Ia, Mb) = dreg1   | |, PM(Ic, Md) = dreg2   |;
          |, dreg1 = DM(Ia, Mb)   | |, dreg2 = PM(Ic, Md)   |
```

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

```
compute   |, DM(Ia+1, 0) = cdreg1   | |, PM(Ic+1, 0) = cdreg2   |;
          |, cdreg1 = DM(Ia+1, 0)   | |, cdreg2 = PM(Ic+1, 0)   |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
R7=BSET R6 BY R0, DM(I0,M3)=R5, PM(I11,M15)=R4;
R8=DM(I4,M1), PM(I12 M12)=R0;
```

When the processor is in SISD mode, the first instruction in this example performs a computation along with two memory writes. DAG1 is used to write to DM and DAG2 is used to write to PM. In the second instruction, a read from data memory to register `R8` and a write to program memory from register `R0` are performed.

When the ADSP-2136x processor is in SIMD mode, the first instruction in this example performs the same computation and performs two writes in parallel on both PEx and PEy. The `R7` register on PEx and `S7` on PEy both store the results of the Bset computations. Also, simultaneous dual memory writes occur with DM and PM, writing in values from `R5`, `S5`

(DM) and `R4`, `S4` (PM) respectively. In the second instruction, values are simultaneously read from data memory to registers `R8` and `S8` and written to program memory from registers `R0` and `S0`.

```
R0=DM(I1,M1);
```

When the processor is in broadcast mode (the `BDCST1` bit is set in the `MODE1` system register), the `R0` (PEx) data register in this example is loaded with the value from data memory utilizing the `I1` register from DAG1, and `S0` (PEy) is loaded with the same value.

Type 1 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
001			DMD	DMI			DMM			PMD	DM DREG				PMI			PMM			PM DREG			

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Bits	Description
DMD, PMD	Select the access types (read or write)
DM DREG, PM DREG	Specify register file location
DMI, PMI	Specify I registers for data and program memory
DMM, PMM	Specify M registers used to update the I registers
COMPUTE	Defines a compute operation to be performed in parallel with the data accesses; if omitted, this is a NOP

Type 2: Compute

Compute operation, optional condition

Syntax

```
IF  COND compute  ;
```

Function (SISD)

In SISD mode, the Type 2 instruction provides a conditional `compute` instruction. The instruction is executed if the specified `condition` tests true.

Function (SIMD)

In SIMD mode, the Type 2 instruction provides the same conditional `compute` instruction as is available in SISD mode, but provides the operation simultaneously for the X and Y processing elements. The instruction is executed in a processing element if the specified `condition` tests true in that element independent of the `condition` result for the other element.

The following pseudo code compares the Type 2 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

```
IF  PEx COND compute  ;
```

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

```
IF  PEy COND compute  ;
```

Do not use the pseudo code above as instruction syntax.

Examples

```
IF MV R6=SAT MRF (UI);
```

When the ADSP-2136x processor is in SISD, the condition is evaluated in the PEx processing element. If the condition is true, the computation is performed and the result is stored in register `R6`.

When the processor is in SIMD mode, the condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PEs, either one PE, or neither PE dependent on the outcome of the condition. If the condition is true in PEx, the computation is performed and the result is stored in register `R6`. If the condition is true in PEy, the computation is performed and the result is stored in register `S6`.

Type 2 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			00001							COND														

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Bits	Description
COND	Selects whether the operation specified in the COMPUTE field is executed. If the COND is true, the compute is executed. If no condition is specified, COND is TRUE condition, and the compute is executed.

Type 3: Compute, ureg«··»DM | PM, register modify

Transfer operation between data or program memory and universal register, optional condition, optional compute operation

Syntax

```
IF COND compute     |, DM(Ia, Mb)     |=  ureg (LW);
                    |, PM(Ic, Md)     |

                    |, DM(Mb, Ia)     |=  ureg (LW);
                    |, PM(Md, Ic)     |

                     , ureg =         |DM(Ia, Mb) (LW);    |
                                      |PM(Ic, Md) (LW);    |

                     , ureg =         |DM(Mb, Ia) (LW);    |
                                      |PM(Md, Ic) (LW);    |
```

Function (SISD)

In SISD mode, the Type 3 instruction provides access between data or program memory and a universal register. The specified I register addresses data or program memory. The I value is either pre-modified (M, I order) or post-modified (I, M order) by the specified M register. If it is post-modified, the I register is updated with the modified value. If a `compute` operation is specified, it is performed in parallel with the data access. The optional `(LW)` in this syntax lets programs specify long word addressing, overriding default addressing from the memory map. If a `condition` is specified, it affects the entire instruction. Note that the *`Ureg`* may not be from the same DAG (that is, DAG1 or DAG2) as `Ia/Mb` or `Ic/Md`. For more information on register restrictions, see "Data Address Generators" in Chapter 4, Data Address Generators.

Function (SIMD)

In SIMD mode, the Type 3 instruction provides the same access between data or program memory and a universal register as is available in SISD mode, but provides this operation simultaneously for the X and Y processing elements.

The X element uses the specified I register to address data or program memory. The I value is either pre-modified (M, I order) or post-modified (I, M order) by the specified M register. The Y element adds one to the specified I register (before pre-modify or post-modify) to address data or program memory. If the broadcast read bits—`BDCST1` (for `I1`) or `BDCST9` (for `I9`)—are set, the Y element uses the specified I and M registers without adding one. If the I value post-modified, the I register is updated with the modified value from the specified M register. The optional `(LW)` in this syntax lets programs specify long word addressing, overriding default addressing from the memory map.

For the universal register, the X element uses the specified *`Ureg`* register, and the Y element uses the corresponding complementary register (*`Cureg`*). For a list of complementary registers, see Table A-10 on page A-25. Note that the *`Ureg`* may not be from the same DAG (DAG1 or DAG2) as `Ia/Mb` or `Ic/Md`.

If a `compute` operation is specified, it is performed simultaneously on the X and Y processing elements in parallel with the data access. If a `condition` is specified, it affects the entire instruction. The instruction is executed in a processing element if the specified `condition` tests true in that element independent of the `condition` result for the other element.

The following pseudo code compares the Type 3 instruction's explicit and implicit operations in SIMD mode.

```
SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)
IF PEx COND compute   |, DM(Ia, Mb)        |=  ureg (LW);
                      |, PM(Ic, Md)        |

                      |, DM(Mb, Ia)        |=  ureg (LW);
                      |, PM(Md, Ic)        |

                       , ureg =            |DM(Ia, Mb) (LW);    |
                                           |PM(Ic, Md) (LW);    |

                       , ureg =            |DM(Mb, Ia) (LW);    |
                                           |PM(Md, Ic) (LW);    |

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)
IF PEy COND compute   |, DM(Ia+1, 0)       |=  cureg (LW);
                      |, PM(Ic+1, 0)       |

                      |, DM(Mb+1, Ia)      |=  cureg (LW);
                      |, PM(Md+1, Ic)      |

                       , cureg =           |DM(Ia+1, 0) (LW);   |
                                           |PM(Ic+, 0) (LW);    |

                       , cureg =           |DM(Mb+1, Ia) (LW);  |
                                           |PM(Md+1, Ic) (LW);  |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
R6=R3-R11, DM(I0,M1)=ASTATx;
IF NOT SV F8=CLIP F2 BY F14, F7=PM(I12,M12);
```

When the processor is in SISD mode, the computation and a data memory write in the first instruction are performed in PEx. The second instruction stores the result of the computation in `F8`, and the result of the program memory read into `F7` if the condition's outcome is true.

When the ADSP-2136x processor is in SIMD, the result of the computation in PEx in the first instruction is stored in `R6`, and the result of the parallel computation in PEy is stored in `S6`. In addition, there is a simultaneous data memory write of the values stored in `ASTATx` and `ASTATy`. The condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PEs, either one PE, or neither PE, dependent on the outcome of the condition. If the condition is true in PEx, the computation is performed, the result is stored in register `F8` and the result of the program memory read is stored in `F7`. If the condition is true in PEy, the computation is performed, the result is stored in register `SF8`, and the result of the program memory read is stored in `SF7`.

```
IF NOT SV F8=CLIP F2 BY F14, F7=PM(I9,M12);
```

When the ADSP-2136x processor is in broadcast mode (the `BDCST9` bit is set in the `MODE1` system register) and the condition tests true, the computation is performed and the result is stored in register `F8`. Also, the result of the program memory read via the `I9` register from DAG2 is stored in `F7`. The `SF7` register is loaded with the same value from program memory as `F7`.

Type 3: Compute, ureg«··»DM | PM, register modify

Type 3 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
010			U	I			M			COND					G	D	L	UREG						

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Bits	Description
COND	Specifies the test condition; if omitted, COND is TRUE
D	Selects the access Type (read or write)
G	Selects data memory or program memory
L	Forces a long word (LW) access when address is in normal word address range
UREG	Specifies the universal register
I	Specifies the I register
M	Specifies the M register
U	Selects either update (post-modify) or no update (pre-modify)
COMPUTE	Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP

Type 4: Compute, dreg«··»DM | PM, data modify

PC-relative transfer between data or program memory and register file, optional condition, optional compute operation

Syntax

```
IF COND compute         |, DM(Ia, <data6>)      | = dreg ;
                        |, PM(Ic, <data6>)      |

                        |, DM(<data6>, Ia)      |= dreg ;
                        |, PM(<data6>, Ic)      |

                         , dreg  =              |DM(Ia, <data6>)  ;      |
                                                |PM(Ic, <data6>)  ;      |

                         , dreg  =              |DM(<data6>, Ia)  ;      |
                                                |PM(<data6>, Ic)  ;      |
```

Function (SISD)

In SISD mode, the Type 4 instruction provides access between data or program memory and the register file. The specified I register addresses data or program memory. The I value is either pre-modified (data order, I) or post-modified (I, data order) by the specified immediate data. If it is post-modified, the I register is updated with the modified value. If a `compute` operation is specified, it is performed in parallel with the data access. If a `condition` is specified, it affects the entire instruction. For more information on register restrictions, see “Data Address Generators” in Chapter 4, Data Address Generators.

Function (SIMD)

In SIMD mode, the Type 4 instruction provides the same access between data or program memory and the register file as is available in SISD mode, but provides the operation simultaneously for the X and Y processing elements.

The X element uses the specified I register to address data or program memory. The I value is either pre-modified (data, I order) or post-modified (I, data order) by the specified immediate data. The Y element adds one to the specified I register (before pre-modify or post-modify) to address data or program memory. If the broadcast read bits—`BDCST1` (for `I1`) or `BDCST9` (for `I9`)—are set, the Y element uses the specified I and M registers without adding one. If the I value post-modified, the I register is updated with the modified value from the specified M register. The optional `(LW)` in this syntax lets programs specify long word addressing, overriding default addressing from the memory map.

For the data register, the X element uses the specified *`Dreg`* register, and the Y element uses the corresponding complementary register (*`Cdreg`*). For a list of complementary registers, see Table A-10 on page A-25.

If a `compute` operation is specified, it is performed simultaneously on the X and Y processing elements in parallel with the data access. If a `condition` is specified, it affects the entire instruction, not just the computation. The instruction is executed in a processing element if the specified `condition` tests true in that element independent of the `condition` result for the other element.

The following pseudo code compares the Type 4 instruction's explicit and implicit operations in SIMD mode.

```
SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)
IF PEx COND compute   |, DM(Ia, <data6>)       | = dreg ;
                      |, PM(Ic, <data6>)       |

                      |, DM(<data6>, Ia)       |= dreg ;
                      |, PM(<data6>, Ic)       |

                      , dreg  =                |DM(Ia, <data6>)  ;         |
                                               |PM(Ic, <data6>)  ;         |

                      , dreg  =                |DM(<data6>, Ia)  ;         |
                                               |PM(<data6>, Ic)  ;         |

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)
IF PEy COND compute   |, DM(Ia+1, 0)           | = cdreg ;
                      |, PM(Ic+1, 0)           |

                      |, DM(<data6>+1, Ia)     |= cdreg ;
                      |, PM(<data6>+1, Ic)     |

                      , cdreg  =               |DM(Ia+1, 0)  ;             |
                                               |PM(Ic+1, 0)  ;             |

                      , cdreg  =               |DM(<data6>+1, Ia)  ;       |
                                               |PM(<data6>+1, Ic)  ;       |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
IF FLAG0_IN F1=F5*F12, F11=PM(I10,6);
R12=R3 AND R1, DM(6,I1)=R6;
```

When the processor is in SISD mode, the computation and program memory read in the first instruction are performed in PEx if the condition's outcome is true. The second instruction stores the result of the logical AND in R12 and writes the value within R6 into data memory.

When the processor is in SIMD mode, the condition is evaluated on each processing element, PEx and PEy, independently. The computation and program memory read execute on both PEs, either one PE, or neither PE dependent on the outcome of the condition. If the condition is true in PEx, the computation is performed, and the result is stored in register F1, and the program memory value is read into register F11. If the condition is true in PEy, the computation is performed, the result is stored in register SF1, and the program memory value is read into register SF11.

```
If FLAG0_IN F1=F5*F12, F11=PM(I9,3);
```

When the ADSP-2136x processor is in broadcast mode (the BDCST9 bit is set in the MODE1 system register) and the condition tests true, the computation is performed, the result is stored in register F1, and the program memory value is read into register F11 via the I9 register from DAG2. The SF11 register is also loaded with the same value from program memory as F11.

Type 4 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
011			0	I			G	D	U	COND					DATA						DREG			

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Bits	Description
COND	Specifies the test condition; if omitted, COND is TRUE
D	Selects the access Type (read or write)
G	Selects data memory or program memory
DREG	Specifies the register file location
I	Specifies the I register
DATA	Specifies a 6-bit, twos-complement modify value
U	Selects either pre-modify without update or post-modify with update
COMPUTE	Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP

Type 5: Compute, ureg«···»ureg | Xdreg<->Ydreg

Transfer between two universal registers or swap between a data register in each processing element, optional condition, optional compute operation

Syntax

```
IF COND compute,      |ureg1 = ureg2        | ;

                      |X dreg <-> Y dreg    |
```

Function (SISD)

In SISD mode, the Type 5 instruction provides transfer (=) from one universal register to another or provides a swap (<->) between a data register in the X processing element and a data register in the Y processing element. If a `compute` operation is specified, it is performed in parallel with the data access. If a `condition` is specified, it affects the entire instruction.

Function (SIMD)

In SIMD mode, the Type 5 instruction provides the same transfer (=) from one register to another as is available in SISD mode, but provides this operation simultaneously for the X and Y processing elements. The swap (<->) operation does the same operation in SISD and SIMD modes; no extra swap operation occurs in SIMD mode.

In the transfer (=), the X element transfers between the universal registers *Ureg*1 and *Ureg*2, and the Y element transfers between the complementary universal registers *Cureg*1 and *Cureg*2. For a list of complementary registers, see Table A-10 on page A-25.

If a `compute` operation is specified, it is performed simultaneously on the X and Y processing elements in parallel with the transfer. If a `condition` is specified, it affects the entire instruction. The instruction is executed in a processing element if the specified `condition` tests true in that element independent of the `condition` result for the other element.

The following pseudo code compares the Type 5 instruction's explicit and implicit operations in SIMD mode.

```
SIMD Explicit Operation (PEx Operation Stated in the Instruction Syntax)
IF PEx COND compute,      |ureg1 = ureg2              | ;

                          |X dreg <-> Y dreg          |

SIMD Implicit Operation (PEy Operation Implied by the Instruction Syntax)
IF PEy COND compute,      |cureg1 = cureg2            | ;

                          |{no implicit operation}    |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
IF TF MRF=R2*R6(SSFR), M4=R0;
LCNTR=L7;
R0 <-> S1;
```

When the ADSP-2136x processor is in SISD mode, the condition in the first instruction is evaluated in the PEx processing element. If the condition is true, `MRF` is loaded with the result of the computation and a register transfer occurs between `R0` and `M4`. The second instruction initializes the loop counter independent of the outcome of the first instruction's condition. The third instruction swaps the register contents between `R0` and `S1`.

When the ADSP-2136x processor is in SIMD mode, the condition is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PEs, either one PE, or neither PE dependent on the outcome of the condition. For the register transfer to complete, the condition must be satisfied in both PEx and PEy. The second instruction initializes the loop counter independent of the outcome of the first instruction's condition. The third instruction swaps the register contents between `R0` and `S1`—the SISD and SIMD swap operation is the same.

Type 5 Opcode (Ureg = Ureg transfer)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
011			1	0	SRC UREG					COND					SU			DEST UREG						

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Type 5 Opcode (X Dreg <-> Y Dreg swap)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
011			1	1		Y DREG				COND											X DREG			

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Bits	Description
COND	Specifies the test condition; if omitted, COND is TRUE
SRC UREG	Identifies the universal register source, (highest 5 bits of register code)
SU	Identifies the universal register source, (lowest 2 bits of register code)
DEST UREG	Identifies the universal register destination
Y DREG	Identifies the PEy data registers for swap (must appear to right of swap operator)
X DREG	Identifies the PEx data register for swap (must appear to left of swap operator)
COMPUTE	Defines a compute operation to be performed in parallel with the data transfer; if omitted, this is a NOP

Type 6: Immediate Shift, dreg«··»DM | PM

Immediate shift operation, optional condition, optional transfer between data or program memory and register file

Syntax

```
IF COND shiftimm          | , DM(Ia, Mb)          | = dreg ;
                          | , PM(Ic, Md)          |

                            , dreg =              | DM(Ia, Mb) ;   |
                                                  | PM(Ic, Md) ;   |
```

Function (SISD)

In SISD mode, the Type 6 instruction provides an immediate shift, which is a shifter operation that takes immediate data as its Y-operand. The immediate data is one 8-bit value or two 6-bit values, depending on the operation. The X-operand and the result are register file locations.

For more information on shifter operations, see "Shifter Operations" on page 9-62. For more information on register restrictions, see "Data Address Generators" in Chapter 4, Data Address Generators.

If an access to data or program memory from the register file is specified, it is performed in parallel with the shifter operation. The I register addresses data or program memory. The I value is post-modified by the specified M register and updated with the modified value. If a `condition` is specified, it affects the entire instruction.

Function (SIMD)

In SIMD mode, the Type 6 instruction provides the same immediate shift operation as is available in SISD mode, but provides this operation simultaneously for the X and Y processing elements.

If an access to data or program memory from the register file is specified, it is performed simultaneously on the X and Y processing elements in parallel with the shifter operation.

The X element uses the specified I register to address data or program memory. The I value is post-modified by the specified M register and updated with the modified value.The Y element adds one to the specified I register to address data or program memory. If the broadcast read bits—`BDCST1` (for `I1`) or `BDCST9` (for `I9`)—are set, the Y element uses the specified I and M registers without adding one.

If a `condition` is specified, it affects the entire instruction. The instruction is executed in a processing element if the specified `condition` tests true in that element independent of the `condition` result for the other element.

The following pseudo code compares the Type 6 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

```
IF PEx COND shiftimm    |, DM(Ia, Mb)     |  = dreg  ;
                        |, PM(Ic, Md)     |

                         , dreg  =        |DM(Ia, Mb)  ;    |
                                          |PM(Ic, Md)  ;    |
```

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

```
IF PEy COND shiftimm    |, DM(Ia+1, 0)    |  = cdreg  ;
                        |, PM(Ic+1, 0)    |

                         , cdreg  =       |DM(Ia+1, 0)  ;   |
                                          |PM(Ic+1, 0)  ;   |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
IF GT R2 = LSHIFT R6 BY 0x4, DM(I4,M4)=R0;
IF NOT SZ R3 = FEXT R1 BY 8:4;
```

When the ADSP-2136x processor is in SISD mode, the computation and data memory write in the first instruction are performed in PEx if the condition's outcome is true. In the second instruction, register R3 is loaded with the result of the computation if the outcome of the condition is true.

When the processor is in SIMD mode, the condition is evaluated on each processing element, PEx and PEy, independently. The computation and data memory write executes on both PEs, either one PE, or neither PE dependent on the outcome of the condition. If the condition is true in PEx, the computation is performed, the result is stored in register R2, and the data memory value is written from register R0. If the condition is true in PEy, the computation is performed, the result is stored in register S2, and the value within S0 is written into data memory. The second instruction's condition is also evaluated on each processing element, PEx and PEy, independently. If the outcome of the condition is true, register R3 is loaded with the result of the computation on PEx, and register S3 is loaded with the result of the computation on PEy.

```
R2 = LSHIFT R6 BY 0x4, F3=DM(I1,M3);
```

When the processor is in broadcast mode (the BDCST1 bit is set in the MODE1 system register), the computation is performed, the result is stored in R2, and the data memory value is read into register F3 via the I1 register from DAG1. The SF3 register is also loaded with the same value from data memory as F3.

Type 6 Opcode (with data access)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
100			0	I			M			COND					G	D	DATAEX				DREG			

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	SHIFTOP						DATA								RN				RX			

Type 6 Opcode (without data access)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			00010							COND							DATAEX							

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	SHIFTOP						DATA								RN				RX			

Bits	Description
COND	Specifies the test condition; if omitted, COND is TRUE
SHIFTOP	Specifies the shifter operation. For more information, see "Shifter Operations" on page 9-62
DATA	Specifies an 8-bit immediate shift value. For shifter operations requiring two 6-bit values (a shift value and a length value), the DATAEX field adds 4 MSBs to the DATA field, creating a 12-bit immediate value. The six LSBs are the shift value, and the six MSBs are the length value.
D	Selects the access Type (read or write) if a memory access is specified
G	Selects data memory or program memory

Bits	Description
DREG	Specifies the register file location
I	Specifies the I register, which is post-modified and updated by the M register
M	Identifies the M register for post-modify

Type 7: Compute, modify

Index register modify, optional condition, optional compute operation

Syntax

```
IF COND   compute          , MODIFY          |(Ia, Mb)  ;  |
                                             |(Ic, Md)  ;  |
```

Function (SISD)

In SISD mode, the Type 7 instruction provides an update of the specified I register by the specified M register. If a `compute` operation is specified, it is performed in parallel with the data access. If a `condition` is specified, it affects the entire instruction. For more information on register restrictions, see "Data Address Generators" in Chapter 4, Data Address Generators.

ⓘ If the DAG's `Lx` and `Bx` registers that correspond to `Ia` or `Ic` are set up for circular bufferring, the modify operation always executes circular buffer wraparound, independent of the state of the `CBUFEN` bit.

Function (SIMD)

In SIMD mode, the Type 7 instruction provides the same update of the specified I register by the specified M register as is available in SISD mode, but provides additional features for the optional `compute` operation.

If a `compute` operation is specified, it is performed simultaneously on the X and Y processing elements in parallel with the transfer. If a `condition` is specified, it affects the entire instruction. The instruction is executed in a processing element if the specified `condition` tests true in that element independent of the `condition` result for the other element.

The following pseudo code compares the Type 7 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

```
IF PEx COND   compute      , MODIFY          | (Ia, Mb)  ; |
                                             | (Ic, Md)  ; |
```

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

```
IF PEy COND   compute      {no implied MODIFY operation}
```

Do not use the pseudo code above as instruction syntax.

Examples

```
IF NOT FLAG2_IN R4=R6*R12(SUF), MODIFY(I10,M8);
IF NOT LCE MODIFY(I3,M1);
```

When the processor is in SISD mode, the computation and index register modify in the first instruction are performed in PEx if the condition's outcome is true. In the second instruction, an index register modification occurs if the outcome of the condition is true.

When the ADSP-2136x processor is in SIMD mode, the condition in the first instruction is evaluated on each processing element, PEx and PEy, independently. The computation executes on both PEs, either PE, or neither PE dependent on the outcome of the condition. If the condition is true in PEx, the computation is performed, and the result is stored in `R4`. If the condition is true in PEy, the computation is performed, and the result is stored in `S4`. The index register modify operation occurs based on the logical ORing of the outcome of the conditions tested on both PEs. In the second instruction, the index register modify also occurs based on the logical ORing of the outcomes of the conditions tested on both PEs. Because both threads of a SIMD sequence may be dependent on a single DAG index value, either thread needs to be able to cause a modify of the index.

Type 7 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			00100						G	COND					I			M						

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Bits	Description
COND	Specifies the test condition; if omitted, COND is TRUE
G	Selects DAG1 or DAG2
I	Specifies the I register
M	Specifies the M register
COMPUTE	Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP

Group II Instructions

The instructions in this group contain a `compute` field that specifies a `compute` operation using the ALU, multiplier, or shifter. Because there are a large number of options available for computations, these operations are described separately in "Computations Reference" in Chapter 9, Computations Reference. Note that data moves between the MR registers and the register file are considered multiplier operations and are also covered in "Computations Reference" in Chapter 9, Computations Reference. Group II instructions include the following.

- "Type 8: Direct Jump | Call" on page 8-31

  Direct (or PC-relative) jump/call, optional condition

- "Type 9: Indirect Jump | Call, Compute" on page 8-35

  Indirect (or PC-relative) jump/call, optional condition, optional compute operation

- "Type 10: Indirect Jump | Compute, dreg«···»DM" on page 8-42

  Indirect (or PC-relative) jump or optional compute operation with transfer between data memory and register file

- "Type 11: Return From Subroutine | Interrupt, Compute" on page 8-48

  Return from subroutine or interrupt, optional condition, optional compute operation

- "Type 12: Do Until Counter Expired" on page 8-53

  Load loop counter, do loop until loop counter expired

- "Type 13: Do Until" on page 8-55

  Do until termination

Type 8: Direct Jump | Call

Direct (or PC-relative) jump/call, optional condition

Syntax

```
IF COND JUMP   |<addr24>              |   |(DB)       | ;
               |(PC, <reladdr24>)     |   |(LA)       |
                                          |(CI)       |
                                          |(DB, LA)   |
                                          |(DB, CI)   |

IF COND CALL   |<addr24>              |    (DB) ;
               |(PC, <reladdr24>)     |
```

Function (SISD)

In SISD mode, the Type 8 instruction provides a jump or call to the specified address or PC-relative address. The PC-relative address is a 24-bit, twos-complement value. The Type 8 instruction supports the following modifiers.

- `(DB)`—delayed branch—starts a delayed branch
- `(LA)`—loop abort—causes the loop stacks and PC stack to be popped when the jump is executed. Use the `(LA)` modifier if the jump transfers program execution outside of a loop. Do not use `(LA)` if there is no loop or if the jump address is within the loop.
- `(CI)`—clear interrupt—lets programs reuse an interrupt while it is being serviced

Normally, the ADSP-2136x processor ignores and does not latch an interrupt that reoccurs while its service routine is already executing. Jump (CI) clears the status of the current interrupt without leaving the interrupt service routine, This feature reduces the interrupt routine to a normal subroutine and allows the interrupt to occur again, as a result of a different event or task in the ADSP-2136x processor system. The jump (CI) instruction should be located within the interrupt service routine. For more information on interrupts, see "Program Sequencer" in Chapter 3, Program Sequencer.

To reduce the interrupt service routine to a normal subroutine, the jump (CI) instruction clears the appropriate bit in the interrupt latch register (`IRPTL`) and interrupt mask pointer (`IMASKP`). The ADSP-2136x processor then allows the interrupt to occur again.

When returning from a reduced subroutine, programs must use the (LR) modifier of the RTS if the interrupt occurs during the last two instructions of a loop. For related information, see "Type 11: Return From Subroutine | Interrupt, Compute" on page 8-48.

Function (SIMD)

In SIMD mode, the Type 8 instruction provides the same jump or call operation as in SISD mode, but provides additional features for handling the optional `condition`.

If a `condition` is specified, the jump or call is executed if the specified `condition` tests true in both the X and Y processing elements.

The following pseudo code compares the Type 8 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (Program Sequencer Operation **Stated** in the Instruction Syntax)

IF (PEx AND PEy COND) JUMP	<addr24>		(DB)	;
	(PC, <reladdr24>)		(LA)	
			(CI)	
			(DB, LA)	
			(DB, CI)	

IF (PEx AND PEy COND) CALL	<addr24> (PC, <reladdr24>)	(DB) ;

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

{No explicit PEx operation}

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

{No implicit PEy operation}

Do not use the pseudo code above as instruction syntax.

Examples

```
IF AV JUMP(PC,0x00A4) (LA);
CALL init (DB);   {init is a program label}
JUMP (PC,2) (DB,CI);   {clear current int. for reuse}
```

When the ADSP-2136x processor is in SISD mode, the first instruction performs a jump to the PC-relative address depending on the outcome of the condition tested in PEx. In the second instruction, a jump to the program label `init` occurs. A PC-relative jump takes place in the third instruction.

When the ADSP-2136x processor is in SIMD mode, the first instruction performs a jump to the PC-relative address depending on the logical ANDing of the outcomes of the conditions tested in both PEs. In SIMD mode, the second and third instructions operate the same as in SISD mode. In the second instruction, a jump to the program label `init` occurs. A PC-relative jump takes place in the third instruction.

Type 8: Direct Jump | Call

Type 8 Opcode (with direct branch)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			00110					B	A	COND											J		CI

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDR																							

Type 8 Opcode (with PC-relative branch)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			00111					B	A	COND											J		CI

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RELADDR																							

Bits	Description
COND	Specifies the test condition; if omitted, COND is TRUE
B	Selects the branch type, jump or call; for calls, A and CI are ignored
J	Determines whether the branch is delayed or non-delayed
ADDR	Specifies a 24-bit program memory address
A	Activates loop abort
CI	Activates clear interrupt
RELADDR	Holds a 24-bit, twos-complement value that is added to the current PC value to generate the branch address

Type 9: Indirect Jump | Call, Compute

Indirect (or PC-relative) jump/call, optional condition, optional compute operation

Syntax

```
IF COND JUMP  |(Md, Ic)          | |(DB)      | |, compute        |;
              |(PC, <reladdr6>)  | |(LA)      | |, ELSE compute   |
                                   |(CI)      |
                                   |(DB, LA)  |
                                   |(DB, CI)  |

IF COND CALL  |(Md, Ic)          |  (DB)        |, compute        |;
              |(PC, <reladdr6>)  |              |, ELSE compute   |
```

Function (SISD)

In SISD mode, the Type 9 instruction provides a jump or call to the specified PC-relative address or pre-modified I register value. The PC-relative address is a 6-bit, two's-complement value. If an I register is specified, it is modified by the specified M register to generate the branch address. The I register is not affected by the modify operation. The Type 9 instruction supports the following modifiers:

- `(DB)`—delayed branch—starts a delayed branch
- `(LA)`—loop abort—causes the loop stacks and PC stack to be popped when the jump is executed. Use the (LA) modifier if the jump transfers program execution outside of a loop. Do not use `(LA)` if there is no loop or if the jump address is within the loop.
- `(CI)`—clear interrupt—lets programs reuse an interrupt while it is being serviced

Normally, the ADSP-2136x processor ignores and does not latch an interrupt that reoccurs while its service routine is already executing. Jump (CI) clears the status of the current interrupt without leaving the interrupt service routine. This feature reduces the interrupt routine to a normal subroutine and allows the interrupt to occur again, as a result of a different event or task in the system. The jump (CI) instruction should be located within the interrupt service routine. For more information on interrupts, see "Program Sequencer" in Chapter 3, Program Sequencer.

To reduce an interrupt service routine to a normal subroutine, the jump (CI) instruction clears the appropriate bit in the interrupt latch register (`IRPTL`) and interrupt mask pointer (`IMASKP`). The processor then allows the interrupt to occur again.

When returning from a reduced subroutine, programs must use the (LR) modifier of the RTS instruction if the interrupt occurs during the last two instructions of a loop. For related information, see "Type 11: Return From Subroutine | Interrupt, Compute" on page 8-48.

The jump or call is executed if the optional specified `condition` is true or if no `condition` is specified. If a `compute` operation is specified without the `ELSE`, it is performed in parallel with the jump or call. If a `compute` operation is specified with the `ELSE`, it is performed only if the `condition` specified is false. Note that a `condition` must be specified if an `ELSE compute` clause is specified.

Function (SIMD)

In SIMD mode, the Type 9 instruction provides the same jump or call operation as is available in SISD mode, but provides additional features for the optional `condition`.

If a `condition` is specified, the jump or call is executed if the specified `condition` tests true in both the X and Y processing elements.

If a `compute` operation is specified without the `ELSE`, it is performed by the processing element(s) in which the `condition` test true in parallel with the jump or call. If a `compute` operation is specified with the `ELSE`, it is performed in an element when the `condition` tests false in that element. Note that a `condition` must be specified if an `ELSE compute` clause is specified.

Note that for the `compute`, the X element uses the specified registers and the Y element uses the complementary registers. For a list of complementary registers, see Table 2-16 on page 2-47.

The following pseudo code compares the Type 9 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

IF (PEx AND PEy COND) JUMP	(Md, Ic)	(DB)	, (if PEx COND) compute	;
	(PC, <reladdr6>)	(LA)	, ELSE (if NOT PEx) compute	
		(CI)		
		(DB, LA)		
		(DB, CI)		

IF (PEx AND PEy COND) CALL	(Md, Ic)	(DB)	, (if PEx COND) compute	;
	(PC, <reladdr6>)		, ELSE (if NOT PEx) compute	

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

IF (PEx AND PEy COND) JUMP	(Md, Ic)	(DB)	, (if PEy COND) compute	;
	(PC, <reladdr6>)	(LA)	, ELSE (if NOT PEy) compute	
		(CI)		
		(DB, LA)		
		(DB, CI)		

IF (PEx AND PEy COND) CALL	(Md, Ic)	(DB)	, (if PEy COND) compute	;
	(PC, <reladdr6>)		, ELSE (if NOT PEy) compute	

Do not use the pseudo code above as instruction syntax.

Examples

```
JUMP(M8,I12), R6=R6-1;
IF EQ CALL(PC,17)(DB), ELSE R6=R6-1;
```

When the ADSP-2136x processor is in SISD mode, the indirect jump and compute in the first instruction are performed in parallel. In the second instruction, a call occurs if the condition is true, otherwise the computation is performed.

When the processor is in SIMD mode, the indirect jump in the first instruction occurs in parallel with both processing elements executing computations. In PEx, `R6` stores the result, and `S6` stores the result in PEy. In the second instruction, the condition is evaluated independently on each processing element, PEx and PEy. The call executes based on the logical AND'ing of the PEx and PEy conditional tests. So, the call executes if the condition tests true in both PEx and PEy. Because the `ELSE` inverts the conditional test, the computation is performed independently on either PEx or PEy based on the negative evaluation of the condition code seen by that processing element. If the computation is executed, `R6` stores the result of the computation in PEx, and `S6` stores the result of the computation in PEy.

For a summary of SISD/SIMD conditional testing, see "SISD/SIMD Conditional Testing Summary" on page A-18.

Type 9: Indirect Jump | Call, Compute

Type 9 Opcode (with indirect branch)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			01000					B	A	COND					PMI			PMM			J	E	CI	

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Type 9 Opcode (with PC-relative branch)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			01001					B	A	COND					RELADDR						J	E	CI	

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Bits	Description
COND	Specifies the test condition; if omitted, COND is true
E	Specifies whether or not an ELSE clause is used
B	Selects the branch type, jump or call; for calls, A and CI are ignored
J	Selects delayed or non-delayed branch
A	Activates loop abort
CI	Activates clear interrupt
COMPUTE	Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP

Bits	Description
RELADDR	Holds a 6-bit, twos-complement value that is added to the current PC value to generate the branch address
PMI	Specifies the I register for indirect branches; the I register is pre-modified but not updated by the M register
PMM	Specifies the M register for pre-modifies

Type 10: Indirect Jump | Compute, dreg«··»DM

Indirect (or PC-relative) jump or optional compute operation with transfer between data memory and register file

Syntax

```
IF COND Jump   |(Md, Ic)          |, Else  |compute, DM(Ia, Mb) = dreg ;  |
               |(PC, <reladdr6>)  |        |compute, dreg = DM(Ia, Mb) ;  |
```

Function (SISD)

In SISD mode, the Type 10 instruction provides a conditional jump to either specified PC-relative address or pre-modified I register value. In parallel with the jump, this instruction also provides a transfer between data memory and a data register with optional parallel `compute` operation. For this instruction, the If `condition` and `ELSE` keywords are not optional and must be used. If the specified `condition` is true, the jump is executed. If the specified `condition` is false, the data memory transfer and optional `compute` operation are performed in parallel. Only the `compute` operation is optional in this instruction.

The PC-relative address for the jump is a 6-bit, twos-complement value. If an I register is specified (`Ic`), it is modified by the specified M register (`Md`) to generate the branch address. The I register is not affected by the modify operation. For this jump, programs may not use the delay branch (DB), loop abort (LA), or clear interrupt (CI) modifiers.

For the data memory access, the I register (`Ia`) provides the address. The I register value is post-modified by the specified M register (`Mb`) and is updated with the modified value. Pre-modify addressing is not available for this data memory access.

Function (SIMD)

In SIMD mode, the Type 10 instruction provides the same conditional jump as is available in SISD mode, but the jump is executed if the specified `condition` tests true in both the X or Y processing elements.

In parallel with the jump, this instruction also provides a transfer between data memory and a data register in the X and Y processing elements. An optional parallel `compute` operation for the X and Y processing elements is also available.

For this instruction, the If `condition` and `ELSE` keywords are not optional and must be used. If the specified `condition` is true in both processing elements, the jump is executed. The the data memory transfer and optional `compute` operation specified with the `ELSE` are performed in an element when the `condition` tests false in that element.

Note that for the `compute`, the X element uses the specified *`Dreg`* register and the Y element uses the complementary *`Cdreg`* register. For a list of complementary registers, see Table 2-16 on page 2-47. Only the `compute` operation is optional in this instruction.

The addressing for the jump is the same in SISD and SIMD modes, but addressing for the data memory access differs slightly. For the data memory access in SIMD mode, X processing element uses the specified I register (`Ia`) to address memory. The I register value is post-modified by the specified M register (`Mb`) and is updated with the modified value. The Y element adds one to the specified I register to address memory. Pre-modify addressing is not available for this data memory access.

The following pseudo code compares the Type 10 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

IF (PEx AND PEy COND) Jump	(Md, Ic) (PC, <reladdr6>)	, Else (if NOT PEx)	compute, DM(Ia, Mb) = dreg ; compute, dreg = DM(Ia, Mb) ;

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

IF (PEx AND PEy COND) Jump	(Md, Ic) (PC, <reladdr6>)	, Else (if NOT PEy)	compute, DM(Ia, Mb) = dreg ; compute, dreg = DM(Ia, Mb) ;

Do not use the pseudo code above as instruction syntax.

Examples

```
IF TF JUMP(M8, I8),
ELSE R6=DM(I6, M1);

IF NE JUMP(PC, 0x20),
ELSE F12=FLOAT R10 BY R3, R6=DM(I5, M0);
```

When the processor is in SISD mode, the indirect jump in the first instruction is performed if the condition tests true. Otherwise, R6 stores the value of a data memory read. The second instruction is much like the first, however, it also includes an optional compute, which is performed in parallel with the data memory read.

When the ADSP-2136x processor is in SIMD mode, the indirect jump in the first instruction executes depending on the outcome of the conditional in both processing element. The condition is evaluated independently on each processing element, PEx and PEy. The indirect jump executes based on the logical ANDing of the PEx and PEy conditional tests. So, the indirect jump executes if the condition tests true in both PEx and PEy. The data memory read is performed independently on either PEx or PEy based on the negative evaluation of the condition code seen by that PE.

The second instruction is much like the first instruction. The second instruction, however, includes an optional compute also performed in parallel with the data memory read independently on either PEx or PEy and based on the negative evaluation of the condition code seen by that processing element.

For a summary of SISD/SIMD conditional testing, see "SISD/SIMD Conditional Testing Summary" on page A-18.

```
IF TF JUMP(M8,I8), ELSE R6=DM(I1,M1);
```

When the ADSP-2136x processor is in broadcast mode (the `BDCST1` bit is set in the `MODE1` system register), the instruction performs an indirect jump if the condition tests true. Otherwise, `R6` stores the value of a data memory read via the `I1` register from DAG1. The `S6` register is also loaded with the same value from data memory as `R6`.

Type 10: Indirect Jump | Compute, dreg«··»DM

Type 10 Opcode (with indirect jump)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
110			D	DMI			DMM			COND					PMI			PMM			DREG			

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Type 10 Opcode (with PC-relative jump)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
111			D	DMI			DMM			COND					RELADDR						DREG			

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Bits	Description
COND	Specifies the condition to test; not optional
PMI	Specifies the I register for indirect branches; the I register is premodified, but not updated by the M register
PMM	Specifies the M register for pre-modifies
D	Selects the data memory access Type (read or write)
DREG	Specifies the register file location
DMI	Specifies the I register that is post-modified and updated by the M register
DMM	Identifies the M register for post-modifies
COMPUTE	Defines a compute operation to be performed in parallel with the data access; if omitted, this is a NOP
RELADDR	Holds a 6-bit, twos-complement value that is added to the current PC value to generate the branch address

Type 11: Return From Subroutine | Interrupt, Compute

Indirect (or PC-relative) jump or optional compute operation with transfer between data memory and register file

Syntax

```
IF COND RTS          |(DB)       |   |, compute        |;
                     |(LR)       |   |, ELSE compute   |
                     |(DB, LR)   |

IF COND RTI           (DB)           |, compute        |;
                                     |, ELSE compute   |
```

Function (SISD)

In SISD mode, the Type 11 instruction provides a return from a subroutine (RTS) or return from an interrupt service routine (RTI). A return causes the processor to branch to the address stored at the top of the PC stack. The difference between RTS and RTI is that the RTS instruction only pops the return address off the PC stack, while the RTI does that plus:

- Pops status stack if the `ASTAT` and `MODE1` status registers have been pushed—if the interrupt was `IRQ2-0`, the timer interrupt, or the `VIRPT` vector interrupt
- Clears the appropriate bit in the interrupt latch register (`IRPTL`) and the interrupt mask pointer (`IMASKP`)

The return executes when the optional If `condition` is true (or if no `condition` is specified). If a `compute` operation is specified without the `ELSE`, it is performed in parallel with the return. If a `compute` operation is specified with the `ELSE`, it is performed only when the If `condition` is false. Note that a `condition` must be specified if an `ELSE compute` clause is specified.

RTS supports two modifiers (DB) and (LR); RTI supports one modifier, (DB). If the delayed branch (DB) modifier is specified, the return is delayed; otherwise, it is non-delayed.

If the return is not a delayed branch and occurs as one of the last three instructions of a loop, programs must use the loop reentry (LR) modifier with the subroutine's RTS instruction. The (LR) modifier assures proper reentry into the loop. For example, the processor checks the termination `condition` in counter-based loops by decrementing the current loop counter (`CURLCNTR`) during execution of the instruction two locations before the end of the loop. In this case, the RTS (LR) instruction prevents the loop counter from being decremented again, avoiding the error of decrementing twice for the same loop iteration.

Programs must also use the (LR) modifier for RTS when returning from a subroutine that has been reduced from an interrupt service routine with a jump (CI) instruction. This case occurs when the interrupt occurs during the last two instructions of a loop. For a description of the jump (CI) instruction, see "Type 8: Direct Jump | Call" on page 8-31 or "Type 9: Indirect Jump | Call, Compute" on page 8-35.

Function (SIMD)

In SIMD mode, the Type 11 instruction provides the same return operations as are available in SISD mode, except that the return is executed if the specified `condition` tests true in both the X and Y processing elements.

In parallel with the return, this instruction also provides a parallel `compute` or `ELSE compute` operation for the X and Y processing elements. If a `condition` is specified, the optional `compute` is executed in a processing element if the specified `condition` tests true in that processing element. If a `compute` operation is specified with the `ELSE`, it is performed in an element when the `condition` tests false in that element.

Note that for the `compute`, the X element uses the specified registers, and the Y element uses the complementary registers. For a list of complementary registers, see Table 2-16 on page 2-47.

The following pseudo code compares the Type 11 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

```
IF  (PEx AND PEy COND)  RTS   |(DB)     | , (if PEx COND) compute          | ;
                              |(LR)     | , ELSE (if NOT PEx) compute      |
                              |(DB, LR) |

IF  (PEx AND PEy COND)  RTI    (DB)     | , (if PEx COND) compute          | ;
                                        | , ELSE (if NOT PEx) compute      |
```

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

```
IF  (PEx AND PEy COND)  RTS   |(DB)     | , (if PEy COND) compute          | ;
                              |(LR)     | , ELSE (if NOT PEy) compute      |
                              |(DB, LR) |

IF  (PEx AND PEy COND)  RTI    (DB)     | , (if PEy COND) compute          | ;
                                        | , ELSE (if NOT PEy) compute      |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
RTI, R6=R5 XOR R1;
IF le RTS(DB);
IF sz RTS, ELSE R0=LSHIFT R1 BY R15;
```

When the ADSP-2136x processor is in SISD mode, the first instruction performs a return from interrupt and a computation in parallel. The second instruction performs a return from subroutine only if the condition is true. In the third instruction, a return from subroutine is executed if the condition is true. Otherwise, the computation executes.

When the ADSP-2136x processor is in SIMD mode, the first instruction performs a return from interrupt and both processing elements execute the computation in parallel. The result from PEx is placed in `R6`, and the result from PEy is placed in `S6`. The second instruction performs a return from subroutine (RTS) if the condition tests true in both PEx or PEy. In the third instruction, the condition is evaluated independently on each processing element, PEx and PEy. The RTS executes based on the logical ANDing of the PEx and PEy conditional tests. So, the RTS executes if the condition tests true in both PEx and PEy. Because the `ELSE` inverts the conditional test, the computation is performed independently on either PEx or PEy based on the negative evaluation of the condition code seen by that processing element. The `R0` register stores the result in PEx, and `S0` stores the result in PEy if the computations are executed.

ⓘ For a summary of SISD/SIMD conditional testing, see "SISD/SIMD Conditional Testing Summary" on page A-18.

Type 11: Return From Subroutine | Interrupt, Compute

Type 11 Opcode (return from subroutine)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			01010							COND											J	E	LR	

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Type 11 Opcode (return from interrupt)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23
000			01011							COND											J	E		

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPUTE																						

Bits	Description
COND	Specifies the test condition; if omitted, COND is true
J	Determines whether the return is delayed or non-delayed
E	Specifies whether an ELSE clause is used
COMPUTE	Defines the compute operation to be performed; if omitted, this is a NOP
LR	Specifies whether or not the loop reentry modifier is specified

Type 12: Do Until Counter Expired

Load loop counter, do loop until loop counter expired

Syntax

LCNTR =	<data16>	, DO	<addr24>	UNTIL LCE;
	ureg		(PC, <reladdr24>)	

Function (SISD and SIMD)

In SISD or SIMD modes, the Type 12 instruction sets up a counter-based program loop. The loop counter `LCNTR` is loaded with 16-bit immediate data or from a universal register. The loop start address is pushed on the `PC` stack. The loop end address and the `LCE` termination condition are pushed on the loop address stack. The end address can be either a label for an absolute 24-bit program memory address, or a `PC`-relative 24-bit two's-complement address. The `LCNTR` is pushed on the loop counter stack and becomes the `CURLCNTR` value. The loop executes until the `CURLCNTR` reaches zero.

Examples

```
LCNTR=100, DO fmax UNTIL LCE;   {fmax is a program label}
LCNTR=R12, DO (PC,16) UNTIL LCE;
```

The ADSP-2136x processor (in SISD or SIMD) executes the action at the indicated address for the duration of the loop.

Type 12: Do Until Counter Expired

Type 12 Opcode (with immediate loop counter load)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			01100					DATA															

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RELADDR																							

Type 12 Opcode (with loop counter load from a Ureg)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			01101					0	UREG														

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RELADDR																							

Bits	Description
RELADDR	Specifies the end-of-loop address relative to the DO LOOP instruction address; the assembler also accepts an absolute address and converts the absolute address to the equivalent relative address for coding
DATA	Specifies a 16-bit value to load into the loop counter (LCNTR) for an immediate load
UREG	Specifies a register containing a 16-bit value to load into the loop counter (LCNTR) for a load from an universal register

Type 13: Do Until

Do until termination

Syntax

DO	<addr24> (PC, <reladdr24>)	UNTIL termination ;

Function (SISD)

In SISD mode, the Type 13 instruction sets up a conditional program loop. The loop start address is pushed on the `PC` stack. The loop end address and the `termination` condition are pushed on the loop stack. The end address can be either a label for an absolute 24-bit program memory address or a `PC`-relative, 24-bit twos-complement address. The loop executes until the `termination` condition tests true.

Function (SIMD)

In SIMD mode, the Type 13 instruction provides the same conditional program loop as is available in SISD mode, except that in SIMD mode the loop executes until the `termination` condition tests true in both the X and Y processing elements.

The following pseudo code compares the Type 13 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (Program Sequencer Operation **Stated** in the Instruction Syntax

```
DO        |<addr24>            |UNTIL (PEx AND PEy) termination ;
          |(PC, <reladdr24>)   |
```

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)
{No explicit PEx operation}

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)
{No implicit PEy operation}

Do not use the pseudo code above as instruction syntax.

Examples

```
DO end UNTIL FLAG1_IN;   {end is a program label}
DO (PC,7) UNTIL AC;
```

When the processor is in SISD mode, the `end` program label in the first instruction specifies the start address for the loop, and the loop is executed until the instruction's condition tests true. In the second instruction, the start address is given in the form of a PC-relative address. The loop executes until the instruction's condition tests true.

When the ADSP-2136x processor is in SIMD mode, the `end` program label in the first instruction specifies the start address for the loop, and the loop is executed until the instruction's condition tests true in both PEx or PEy. In the second instruction, the start address is given in the form of a PC-relative address. The loop executes until the instruction's condition tests true in both PEx or PEy.

Type 13 Opcode (relative addressing)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			01110							TERM													

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RELADDR																							

Bits	Description
RELADDR	Specifies the end-of-loop address relative to the Do Loop instruction address; the assembler accepts an absolute address as well and converts the absolute address to the equivalent relative address for coding
TERM	Specifies the termination condition

Group III Instructions

Group III instructions include the following.

- “Type 14: Ureg«···»DM | PM (direct addressing)” on page 8-59

  Transfer between data or program memory and universal register, direct addressing, immediate address

- “Type 15: Ureg«···»DM | PM (indirect addressing)” on page 8-62

  Transfer between data or program memory and universal register, indirect addressing, immediate modifier

- “Type 16: Immediate data···»DM | PM” on page 8-66

Immediate data write to data or program memory

- “Type 17: Immediate data···»Ureg” on page 8-69

  Immediate data write to universal register

Type 14: Ureg«··»DM | PM (direct addressing)

Transfer between data or program memory and universal register, direct addressing, immediate address

Syntax

```
|DM(<addr32>)            |=  ureg (LW);
|PM(<addr32>)            |

 ureg  =                 |DM(<addr32>) (LW);      |
                         |PM(<addr32>) (LW);      |
```

Function (SISD)

In SISD mode, the Type 14 instruction sets up an access between data or program memory and a universal register, with direct addressing. The entire data or program memory address is specified in the instruction. Addresses are 32 bits wide (0 to 2^{32}–1). The optional (LW) in this syntax lets programs specify long word addressing, overriding default addressing from the memory map.

Function (SIMD)

In SIMD mode, the Type 14 instruction provides the same access between data or program memory and a universal register, with direct addressing, as is available in SISD mode, except that addressing differs slightly, and the transfer occurs in parallel for the X and Y processing elements.

For the memory access in SIMD mode, the X processing element uses the specified 32-bit address to address memory. The Y element adds one to the specified 32-bit address to address memory.

For the universal register, the X element uses the specified *Ureg*, and the Y element uses the complementary register (*Cureg*) that corresponds to the *Ureg* register specified in the instruction. For a list of complementary registers, see Table 2-16 on page 2-47. Note that only the *Cureg* subset registers which have complementary registers are effected by SIMD mode.

The following pseudo code compares the Type 14 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

```
|DM(<addr32>)          |= ureg (LW);
|PM(<addr32>)          |

ureg =                 |DM(<addr32>) (LW);    |
                       |PM(<addr32>) (LW);    |
```

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

```
|DM(<addr32>+1)        |= cureg (LW);
|PM(<addr32>+1)        |

cureg =                |DM(<addr32>+1) (LW);  |
                       |PM(<addr32>+1) (LW);  |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
DM(temp)=MODE1;   {temp is a program label}
WAIT=PM(0x489060);
```

When the ADSP-2136x processor is in SISD mode, the first instruction performs a direct memory write of the value in the `MODE1` register into data memory with the data memory destination address specified by the program label, `temp`. The second instruction initializes the `WAIT` register with the value found in the specified address in program memory.

Because of the register selections in this example, these two instructions operate the same in SIMD and SISD mode. The `MODE1` (`SYSCON`) and `WAIT` (IOP) registers are not included in the *`Cureg`* subset, so they do not operate differently in SIMD mode.

Type 14 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			100			G	D	L	UREG							ADDR (upper 8 bits)							

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDR (lower 24 bits)																							

Bits	Description
D	Selects the access Type (read or write)
G	Selects the memory Type (data or program)
L	Forces a long word (LW) access when address is in normal word address range
UREG	Specifies the number of a universal register
ADDR	Contains the immediate address value

Type 15: Ureg«··»DM | PM (indirect addressing)

Transfer between data or program memory and universal register, indirect addressing, immediate modifier

Syntax

```
|DM(<data32>, Ia)        |= ureg              (LW);
|PM(<data32>, Ic)        |

ureg =                   |DM(<data32>, Ia)    |(LW);
                         |PM(<data32>, Ic)    |
```

Function (SISD)

In SISD mode, the Type 15 instruction sets up an access between data or program memory and a universal register, with indirect addressing using I registers. The I register is pre-modified with an immediate value specified in the instruction. The I register is not updated. Address modifiers are 32 bits wide (0 to 2^{32}–1). The *Ureg* may not be from the same DAG (that is, DAG1 or DAG2) as `Ia/Mb` or `Ic/Md`. For more information on register restrictions, see "Data Address Generators" in Chapter 4, Data Address Generators. The optional (LW) in this syntax lets programs specify long word addressing, overriding default addressing from the memory map.

Function (SIMD)

In SIMD mode, the Type 15 instruction provides the same access between data or program memory and a universal register, with indirect addressing using I registers, as is available in SISD mode, except that addressing differs slightly, and the transfer occurs in parallel for the X and Y processing elements.

The X processing element uses the specified I register—pre-modified with an immediate value—to address memory. The Y processing element adds one to the pre-modified I value to address memory. The I register is not updated.

The *Ureg* specified in the instruction is used for the X processing element transfer and may not be from the same DAG (that is, DAG1 or DAG2) as `Ia/Mb` or `Ic/Md`. The Y element uses the complementary register (*Cureg*) that correspond to the *Ureg* register specified in the instruction. For a list of complementary registers, see Table 2-16 on page 2-47. Note that only the *Cureg* subset registers which have complimentary registers are effected by SIMD mode. For more information on register restrictions, see "Data Address Generators" in Chapter 4, Data Address Generators.

The following pseudo code compares the Type 15 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

```
|DM(<data32>, Ia)        |= ureg              (LW);
|PM(<data32>, Ic)        |

ureg  =                  |DM(<data32>, Ia)   |(LW);
                         |PM(<data32>, Ic)   |
```

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

```
|DM(<data32>+1, Ia)      |= cureg             (LW);
|PM(<data32>+1, Ic)      |

cureg  =                 |DM(<data32>+1, Ia) |(LW);
                         |PM(<data32>+1, Ic) |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
DM(24,I5)=TCOUNT;
USTAT1=PM(offs,I13);   {"offs" is a user-defined constant}
```

When the processor is in SISD mode, the first instruction performs a data memory write, using indirect addressing and the *Ureg* timer register, TCOUNT. The DAG1 register I5 is pre-modified with the immediate value of 24. The I5 register is not updated after the memory access occurs. The second instruction performs a program memory read, using indirect addressing and the system register, USTAT1. The DAG2 register I13 is pre-modified with the immediate value of the defined constant, offs. The I13 register is not updated after the memory access occurs.

Because of the register selections in this example, the first instruction in this example operates the same in SIMD and SISD mode. The TCOUNT (timer) register is not included in the *Cureg* subset, and therefore the first instruction operates the same in SIMD and SISD mode.

The second instruction operates differently in SIMD. The USTAT1 (system) register is included in the *Cureg* subset. Therefore, a program memory read—using indirect addressing and the system register, USTAT1 and its complimentary register USTAT2—is performed in parallel on PEx and PEy respectively. The DAG2 register I13 is pre-modified with the immediate value of the defined constant, offs, to address memory on PEx. This same pre-modified value in I13 is skewed by 1 to address memory on PEy. The I13 register is not updated after the memory access occurs in SIMD mode.

Type 15 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
101			G	I			D	L	UREG							DATA (upper 8 bits)							

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA (lower 24 bits)																							

Bits	Description
D	Selects the access Type (read or write)
G	Selects the memory Type (data or program)
L	Forces a long word (LW) access when address is in normal word address range
UREG	Specifies the number of a universal register
DATA	Specifies the immediate modify value for the I register

Type 16: Immediate data ··»DM | PM

Immediate data write to data or program memory

Syntax

```
|DM(Ia, Mb)          | = <data32> ;
|PM(Ic, Md)          |
```

Function (SISD)

In SISD mode, the Type 16 instruction sets up a write of 32-bit immediate data to data or program memory, with indirect addressing. The data is placed in the most significant 32 bits of the 40-bit memory word. The least significant 8 bits are loaded with 0s. The I register is post-modified and updated by the specified M register.

Function (SIMD)

In SIMD mode, the Type 16 instruction provides the same write of 32-bit immediate data to data or program memory, with indirect addressing, as is available in SISD mode, except that addressing differs slightly, and the transfer occurs in parallel for the X and Y processing elements.

The X processing element uses the specified I register to address memory. The Y processing element adds one to the I register to address memory. The I register is post-modified and updated by the specified M register.

The following pseudo code compares the Type 16 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

```
|DM(Ia, Mb)          | = <data32> ;
|PM(Ic, Md)          |
```

Do not use the pseudo code above as instruction syntax.

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

DM(Ia+1, 0)
PM(Ic+1, 0)
	= <data32> ;

Do not use the pseudo code above as instruction syntax.

Examples

```
DM(I4,M0)=19304;
PM(I14,M11)=count;   {count is user-defined constant}
```

When the ADSP-2136x processor is in SISD mode, the two immediate memory writes are performed on PEx. The first instruction writes to data memory and the second instruction writes to program memory. DAG1 and DAG2 are used to indirectly address the locations in memory to which values are written. The `I4` and `I14` registers are post-modified and updated by `M0` and `M11` respectively.

When the processor is in SIMD mode, the two immediate memory writes are performed in parallel on PEx and PEy. The first instruction writes to data memory and the second instruction writes to program memory. DAG1 and DAG2 are used to indirectly address the locations in memory to which values are written. The `I4` and `I14` registers are post-modified and updated by `M0` and `M11` respectively.

Type 16 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
100			1	I			M			G						DATA (upper 8 bits)							

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA (lower 24 bits)																							

Type 16: Immediate data ··›DM | PM

Bits	Description
I	Selects the I register
M	Selects the M register
G	Selects the memory (data or program)
DATA	Specifies the 32-bit immediate data

Type 17: Immediate data ··»Ureg

Immediate data write to universal register

Syntax

```
ureg = <data32> ;
```

Function (SISD)

In SISD mode, the Type 17 instruction writes 32-bit immediate data to a universal register. If the register is 40 bits wide, the data is placed in the most significant 32 bits, and the least significant 8 bits are loaded with 0s.

Function (SIMD)

In SIMD mode, the Type 17 instruction provides the same write of 32-bit immediate data to universal register as is available in SISD mode, but provides parallel writes for the X and Y processing elements.

The X element uses the specified `Ureg`, and the Y element uses the complementary `Cureg`. Note that only the `Cureg` subset registers which have complimentary registers are effected by SIMD mode. For a list of complementary registers, see Table 2-16 on page 2-47.

The following pseudo code compares the Type 17 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

```
ureg = <data32> ;
```

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

```
cureg = <data32> ;
```

Do not use the pseudo code above as instruction syntax.

Examples

```
ASTATx=0x0;
M15=mod1;   {mod1 is user-defined constant}
```

When the ADSP-2136x processor is in SISD mode, the two instructions load immediate values into the specified registers.

Because of the register selections in this example, the second instruction in this example operates the same in SIMD and SISD mode. The `ASTATx` (system) register is included in the *`Cureg`* subset. In the first instruction, the immediate data write to the system register `ASTATx` and its complimentary register `ASTATy` are performed in parallel on PEx and PEy respectively. In the second instruction, the `M15` register is not included in the *`Cureg`* subset. So, the second instruction operates the same in SIMD and SISD mode.

Type 17 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			01111					0	UREG							DATA (upper 8 bits)							

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA (lower 24 bits)																							

Bits	Description
UREG	Specifies the number of a universal register
DATA	Specifies the immediate modify value for the I register

Group IV Instructions

Group IV instructions include the following.

- "Type 18: System Register Bit Manipulation" on page 8-72

  System register bit manipulation
- "Type 19: I Register Modify | Bit-Reverse" on page 8-75

  Immediate I register modify, with or without bit-reverse
- "Type 20: Push, Pop Stacks, Flush Cache" on page 8-78

  Push or Pop of loop and/or status stacks
- "Type 21: Nop" on page 8-80

  No Operation (NOP)
- "Type 22: Idle" on page 8-81

  Idle
- "Type 25: Cjump/Rframe" on page 8-82

  CJUMP/RFRAME (Compiler-generated instruction)

Type 18: System Register Bit Manipulation

System register bit manipulation

Syntax

```
BIT                 |SET           |sreg  <data32>  ;
                    |CLR           |
                    |TGL           |
                    |TST           |
                    |XOR           |
```

Function (SISD)

In SISD mode, the Type 18 instruction provides a bit manipulation operation on a system register. This instruction can set, clear, toggle or test specified bits, or compare (XOR) the system register with a specified data value. In the first four operations, the immediate data value is a mask.

The set operation sets all the bits in the specified system register that are also set in the specified data value. The clear operation clears all the bits that are set in the data value. The toggle operation toggles all the bits that are set in the data value. The test operation sets the bit test flag (`BTF` in `ASTATx/y`) if all the bits that are set in the data value are also set in the system register. The XOR operation sets the bit test flag (`BTF` in `ASTATx/y`) if the system register value is the same as the data value.

For more information on shifter operations, see "Computations Reference" in Chapter 9, Computations Reference. For more information on system registers, see "Control and Status System Registers" on page B-2.

Function (SIMD)

In SIMD mode, the Type 18 instruction provides the same bit manipulation operations as are available in SISD mode, but provides them in parallel for the X and Y processing elements.

The X element operation uses the specified Sreg, and the Y element operations uses the complementary Csreg. For a list of complementary registers, see Table 2-16 on page 2-47.

The following pseudo code compares the Type 18 instruction's explicit and implicit operations in SIMD mode.

SIMD **Explicit** Operation (PEx Operation **Stated** in the Instruction Syntax)

```
BIT          |SET          |sreg  <data32>  ;
             |CLR          |
             |TGL          |
             |TST          |
             |XOR          |
```

SIMD **Implicit** Operation (PEy Operation **Implied** by the Instruction Syntax)

```
BIT          |SET          |csreg  <data32>  ;
             |CLR          |
             |TGL          |
             |TST          |
             |XOR          |
```

Do not use the pseudo code above as instruction syntax.

Examples

```
BIT SET MODE2 0x00000070;
BIT TST ASTATx 0x00002000;
```

When the processor is in SISD mode, the first instruction sets all of the bits in the `MODE2` register that are also set in the data value, bits 4, 5, and 6 in this case. The second instruction sets the bit test flag (`BTF` in `ASTATx`) if all the bits set in the data value, just bit 13 in this case, are also set in the system register.

Because of the register selections in this example, the first instruction operates the same in SISD and SIMD, but the second instruction operates differently in SIMD. Only the *`Cureg`* subset registers which have complimentary registers are affected in SIMD mode. The `ASTATx` (system) register is included in the *`Cureg`* subset, so the bit test operations are performed independently on each processing element in parallel using these complimentary registers. The `BTF` is set on both PEs (`ASTATx` and `ASTATy`), either one PE (`ASTATx` or `ASTATy`), or neither PE dependent on the outcome of the bit test operation.

Type 18 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			10100					BOP				SREG				DATA (upper 8 bits)							

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA (lower 24 bits)																							

Bits	Description
BOP	Selects one of the five bit operations
SREG	Specifies the system register
DATA	Specifies the data value

Type 19: I Register Modify | Bit-Reverse

Immediate I register modify, with or without bit-reverse

Syntax

```
MODIFY          |(Ia, <data32>)  | ;
                |(Ic, <data32>)  |

BITREV          |(Ia, <data32>)  | ;
                |(Ic, <data32>)  |
```

Function (SISD & SIMD)

In SISD and SIMD modes, the Type 19 instruction modifies and updates the specified I register by an immediate 32-bit data value. If the address is to be bit-reversed, programs must specify a DAG1 `Ia` register (`I0`–`I7`) or DAG2 `Ic` register (`I8`–`I15`), and the modified value is bit-reversed before being written back to the I register. No address is output in either case. For more information on register restrictions, see "Data Address Generators" in Chapter 4, Data Address Generators.

If the DAG's `Lx` and `Bx` registers that correspond to `Ia` or `Ic` are set up for circular bufferring, the modify operation always executes circular buffer wraparound, independent of the state of the `CBUFEN` bit.

Examples

```
MODIFY (I4,304);
BITREV (I7,space);   {space is a user-defined constant}
```

In SISD and SIMD, the first instruction modifies and updates the `I4` register by the immediate value of 304. The second instruction utilizes the DAG1 register `I7`. The value originally stored in `I7` is modified by the defined constant, `space`, and is then bit-reversed before being written back to the `I7` register.

Type 19 Opcode (without bit-reverse)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			10110					0	G				I			DATA (upper 8 bits)							

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA (lower 24 bits)																							

Type 19 Opcode (with bit-reverse)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			10110					1	G				I			DATA (upper 8 bits)							

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA (lower 24 bits)																							

Bits	Description

G	Selects the data address generator:
	G=0 for DAG1
	G=1 for DAG2

I	Selects the I register:
	I=0–7 for I0–I7 (for DAG1)
	I=0–7 for I8–I15 (for DAG2)

DATA	Specifies the immediate modifier

Type 20: Push, Pop Stacks, Flush Cache

Push or Pop of loop and/or status stacks

Syntax

```
|PUSH   |LOOP ,  |PUSH   |STS ,   |PUSH   |PCSTK ,  FLUSH CACHE  ;
|POP    |        |POP    |        |POP    |
```

Function (SISD and SIMD)

In SISD and SIMD modes, the Type 20 instruction pushes or pops the loop address and loop counter stacks, the status stack, and/or the PC stack, and/or clear the instruction cache. Any of set of pushes (push loop, push sts, push pcstk) or pops (pop loop, pop sts, pop pcstk) may be combined in a single instruction, but a push may not be combined with a pop.

Flushing the instruction cache invalidates all entries in the cache, with no latency—the cache is cleared at the end of the cycle.

Examples

```
PUSH LOOP, PUSH STS;
POP PCSTK, FLUSH CACHE;
```

In SISD and SIMD, the first instruction pushes the loop stack and status stack. The second instruction pops the PC stack and flushes the cache.

Type 20 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			10111					LPU	LPO	SPU	SPO	PPU	PPO	FC									

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bits	Description
LPU	Pushes the loop stacks
LPO	Pops the loop stacks
SPU	Pushes the status stack
SPO	Pops the status stack
PPU	Pushes the PC stack
PPO	Pops the PC stack
FC	Causes a cache flush

Type 21: Nop

No Operation (NOP)

Syntax

```
NOP ;
```

Function (SISD and SIMD)

In SISD and SIMD modes, the Type 21 instruction provides a null operation; it increments only the fetch address.

Type 21 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			00000					0															

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Type 22: Idle

Idle

Syntax

```
IDLE   ;
```

Function (SISD and SIMD)

In SISD and SIMD modes, the Type 22 instruction executes a NOP and puts the processor in a low power state. The processor remains in the low power state until an interrupt occurs. On return from the interrupt, execution continues at the instruction following the Idle instruction.

Type 22 Opcode

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
000			00000					1															

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Type 25: Cjump/Rframe

Cjump/Rframe (Compiler-generated instruction)

Syntax

```
CJUMP                      |function                  |(DB)  ;
                           |(PC, <reladdr24>)         |

RFRAME  ;
```

Function (SISD and SIMD)

In SISD mode, the Type 25 instruction (cjump) combines a direct or PC-relative jump with register transfer operations that save the frame and stack pointers. The instruction (rframe) also reverses the register transfers to restore the frame and stack pointers.

The Type 25 instruction is only intended for use by a C (or other high-level-language) compiler. Do not use cjump or rframe in assembly programs.

The different forms of this instruction perform the operations listed in Table 8-1.

Table 8-1. Operations Done by Forms of the Type 25 Instruction

Compiler-Generated Instruction[1]	**Operations Performed in SISD Mode**	**Operations Performed in SIMD Mode**
CJUMP label (DB);	JUMP label (DB), R2=I6, I6=I7;	JUMP label (DB), R2=I6, S2=I6, I6=I7;
CJUMP (PC,raddr) (DB);	JUMP (PC,raddr) (DB), R2=I6, I6=I7;	JUMP (PC,raddr) (DB), R2=I6, S2=I6, I6=I7;
RFRAME;	I7=I6, I6=DM(0,I6);	I7=I6, I6=DM(0,I6), I6=DM(1,I6);

1 In this table, raddr indicates a relative 24-bit address.

Type 25a Opcode (with direct branch)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
0001				1000				0000				0100				0000				0000			

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDR																							

Type 25b Opcode (with PC-relative branch)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
0001				1000				0100				0100				0000				0000			

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RELADDR																							

Bits	Description
ADDR	Specifies a 24-bit program memory address for “function”
RELADDR	Specifies a 24-bit, two’s-complement value added to the current PC value to generate the branch address

Type 25: Cjump/Rframe

Type 25c Opcode (RFRAME)

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
0001				1001				0000				0000				0000				0000			

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				0000				0000				0000				0000				0000			

9 COMPUTATIONS REFERENCE

This chapter describes each compute operation in detail, including its assembly language syntax and opcode field. Compute operations execute in the multiplier, the ALU, and the shifter.

Compute Field

The 23-bit compute field is a mini instruction within the ADSP-21xxx instruction. You can specify a value in this field for a variety of compute operations, which include the following.

- Single-function operations involve a single computation unit.
- Multifunction operations specify parallel operation of the multiplier and the ALU or two operations in the ALU.
- The `MR` register transfer is a special type of compute operation used to access the fixed-point accumulator in the multiplier.

For each operation, the assembly language syntax, the function, and the opcode format and contents are specified. For an explanation of the notation and abbreviations, see "Instruction Set Quick Reference" in Appendix A, Instruction Set Quick Reference.

In single-function operations, the compute field of a single-function operation is made up of the following bit fields.

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CU		Opcode								Rn				Rx				Ry			

Bits	Description
CU	Specifies the computation unit for the compute operation, where: 00=ALU, 01=Multiplier, and 10=Shifter
Opcode	Specifies the compute operation
Rn	Specifies register for the compute result
Rx	Specifies register for the compute's x operand
Ry	Specifies register for the compute's y operand

The compute operation (`Opcode`) is executed in the computation unit (`CU`). The x operand and y operand are input from data registers `(Rx` and `Ry`). The compute result goes to a data register (`Rn`). Note that in some shifter operations, the result register (`Rn`) serves as a result destination and as source for a third input operand.

The available compute operations (`Opcode`) appear in Table 9-1 on page 9-3, Table 9-2 on page 9-4, Table 9-3 on page 9-51, Table 9-4 on page 9-52, and Table 9-8 on page 9-63. These tables are organized by computation unit: "ALU Operations" on page 9-3, "Multiplier Operations" on page 9-50, and "Shifter Operations" on page 9-62. Following each table, each compute operation is described in detail.

ALU Operations

This section describes the ALU operations. Table 9-1 and Table 9-2 on page 9-4 summarize the syntax and opcodes for the fixed-point and floating-point ALU operations, respectively.

ALU Fixed-Point Operations

Table 9-1. Fixed-Point ALU Operations

Syntax	Opcode	Reference Page
Rn = Rx + Ry	0000 0001	on page 9-6
Rn = Rx – Ry	0000 0010	on page 9-7
Rn = Rx + Ry + CI	0000 0101	on page 9-8
Rn = Rx – Ry + CI – 1	0000 0110	on page 9-9
Rn = (Rx + Ry)/2	0000 1001	on page 9-10
COMP(Rx, Ry)	0000 1010	on page 9-11
COMPU(Rx, Ry)	0000 1011	on page 9-12
Rn = Rx + CI	0010 0101	on page 9-13
Rn = Rx + CI – 1	0010 0110	on page 9-14
Rn = Rx + 1	0010 1001	on page 9-15
Rn = Rx – 1	0010 1010	on page 9-16
Rn = – Rx	0010 0010	on page 9-17
Rn = ABS Rx	0011 0000	on page 9-18
Rn = PASS Rx	0010 0001	on page 9-19
Rn = Rx AND Ry	0100 0000	on page 9-20
Rn = Rx OR Ry	0100 0001	on page 9-21
Rn = Rx XOR Ry	0100 0010	on page 9-22
Rn = NOT Rx	0100 0011	on page 9-23

Table 9-1. Fixed-Point ALU Operations (Cont'd)

Syntax	Opcode	Reference Page
Rn = MIN(Rx, Ry)	0110 0001	on page 9-24
Rn = MAX(Rx, Ry)	0110 0010	on page 9-25
Rn = CLIP Rx BY Ry	0110 0011	on page 9-26

ALU Floating-Point Operations

Table 9-2. Floating-Point ALU Operations

Syntax	Opcode	Reference Page
Fn = Fx + Fy	1000 0001	on page 9-27
Fn = Fx – Fy	1000 0010	on page 9-28
Fn = ABS (Fx + Fy)	1001 0001	on page 9-29
Fn = ABS (Fx – Fy)	1001 0010	on page 9-30
Fn = (Fx + Fy)/2	1000 1001	on page 9-31
Fn = COMP(Fx, Fy)	1000 1010	on page 9-32
Fn = –Fx	1010 0010	on page 9-33
Fn = ABS Fx	1011 0000	on page 9-34
Fn = PASS Fx	1010 0001	on page 9-35
Fn = RND Fx	1010 0101	on page 9-36
Fn = SCALB Fx BY Ry	1011 1101	on page 9-37
Rn = MANT Fx	1010 1101	on page 9-38
Rn = LOGB Fx	1100 0001	on page 9-39
Rn = FIX Fx BY Ry	1101 1001	on page 9-40
Rn = FIX Fx	1100 1001	on page 9-40
Rn = TRUNC Fx BY Ry	1101 1101	on page 9-40
Rn = TRUNC Fx	1100 1101	on page 9-40

Table 9-2. Floating-Point ALU Operations (Cont'd)

Syntax	Opcode	Reference Page
Fn = FLOAT Rx BY Ry	1101 1010	on page 9-42
Fn = FLOAT Rx	1100 1010	on page 9-42
Fn = RECIPS Fx	1100 0100	on page 9-43
Fn = RSQRTS Fx	1100 0101	on page 9-45
Fn = Fx COPYSIGN Fy	1110 0000	on page 9-47
Fn = MIN(Fx, Fy)	1110 0001	on page 9-48
Fn = MAX(Fx, Fy)	1110 0010	on page 9-49
Fn = CLIP Fx BY Fy	1110 0011	on page 9-50

Rn = Rx + Ry

Function

Adds the fixed-point fields in registers Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in `MODE1` set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Cleared
AI	Cleared

Rn = Rx – Ry

Function

Subtracts the fixed-point field in register Ry from the fixed-point field in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in `MODE1` set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Cleared
AI	Cleared

Rn = Rx + Ry + CI

Function

Adds with carry (`AC` from `ASTAT`) the fixed-point fields in registers Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in `MODE1` set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Cleared
AI	Cleared

Rn = Rx – Ry + CI – 1

Function

Subtracts with borrow (AC – 1 from ASTAT) the fixed-point field in register Ry from the fixed-point field in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Cleared
AI	Cleared

Rn = (Rx + Ry)/2

Function

Adds the fixed-point fields in registers Rx and Ry and divides the result by 2. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in the `MODE1` register.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Cleared
AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Cleared
AI	Cleared

COMP(Rx, Ry)

Function

Compares the fixed-point field in register Rx with the fixed-point field in register Ry. Sets the `AZ` flag if the two operands are equal, and the `AN` flag if the operand in register Rx is smaller than the operand in register Ry.

The `ASTAT` register stores the results of the previous eight ALU compare operations in bits 24–31. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed. The MSB of `ASTAT` is set if the X operand is greater than the Y operand (its value is the AND of $\overline{\texttt{AZ}}$ and $\overline{\texttt{AN}}$); it is otherwise cleared.

Status Flags

AZ	Set if the operands in registers Rx and Ry are equal, otherwise cleared
AU	Cleared
AN	Set if the operand in the Rx register is smaller than the operand in the Ry register, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Cleared

COMPU(Rx, Ry)

Function

Compares the fixed-point field in register Rx with the fixed-point field in register Ry, Sets the AZ flag if the two operands are equal, and the AN flag if the operand in register Rx is smaller than the operand in register Ry. This operation performs a magnitude comparison of the fixed-point contents of Rx and Ry.

The ASTAT register stores the results of the previous eight ALU compare operations in bits 24–31. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed. The MSB of ASTAT is set if the X operand is greater than the Y operand (its value is the AND of $\overline{\text{AZ}}$ and $\overline{\text{AN}}$); it is otherwise cleared.

Status Flags

AZ	Is set if the operands in registers Rx and Ry are equal, otherwise cleared
AU	Is cleared
AN	Is set if the operand in the Rx register is smaller than the operand in the Ry register, otherwise cleared
AV	Is cleared
AC	Is cleared
AS	Is cleared
AI	Is cleared

Rn = Rx + CI

Function

Adds the fixed-point field in register Rx with the carry flag from the ASTAT register (AC). The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF).

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Cleared
AI	Cleared

Rn = Rx + CI – 1

Function

Adds the fixed-point field in register Rx with the borrow from the `ASTAT` register (`AC` – 1). The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in `MODE1` set) positive overflows return the maximum positive number (0x7FFF FFFF).

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Cleared
AI	Cleared

Rn = Rx + 1

Function

Increments the fixed-point operand in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in `MODE1` set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Set if the XOR of the carries of the two most significant adder, stages is 1, otherwise cleared
AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Cleared
AI	Cleared

Rn = Rx – 1

Function

Decrements the fixed-point operand in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in `MODE1` set), underflow causes the minimum negative number (0x8000 0000) to be returned.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Cleared
AI	Cleared

Rn = –Rx

Function

Negates the fixed-point operand in Rx by two's-complement. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. Negation of the minimum negative number (0x8000 0000) causes an overflow. In saturation mode (the ALU saturation mode bit in `MODE1` set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

Status Flags

AZ	Set if the fixed-point output is all 0s
AU	Cleared
AN	Set if the most significant output bit is 1
AV	Set if the XOR of the carries of the two most significant adder stages is 1
AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Cleared
AI	Cleared

Rn = ABS Rx

Function

Determines the absolute value of the fixed-point operand in Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. The ABS of the minimum negative number (0x8000 0000) causes an overflow. In saturation mode (the ALU saturation mode bit in `MODE1` set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC	Set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Set if the fixed-point operand in Rx is negative, otherwise cleared
AI	Cleared

Rn = PASS Rx

Function

Passes the fixed-point operand in Rx through the ALU to the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Cleared

Rn = Rx AND Ry

Function

Logically ANDs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Cleared

Rn = Rx OR Ry

Function

Logically ORs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Cleared

Rn = Rx XOR Ry

Function

Logically XORs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Cleared

Rn = NOT Rx

Function

Logically complements the fixed-point operand in Rx. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Cleared

Rn = MIN(Rx, Ry)

Function

Returns the smaller of the two fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Cleared

Rn = MAX(Rx, Ry)

Function

Returns the larger of the two fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Cleared

Rn = CLIP Rx BY Ry

Function

Returns the fixed-point operand in Rx if the absolute value of the operand in Rx is less than the absolute value of the fixed-point operand in Ry. Otherwise, returns |Ry| if Rx is positive, and –|Ry| if Rx is negative. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status Flags

AZ	Set if the fixed-point output is all 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Cleared

Fn = Fx + Fy

Function

Adds the floating-point operands in registers Fx and Fy. The normalized result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in `MODE1`. Post-rounded overflow returns ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±zero. Denormal inputs are flushed to ±zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Set if the post-rounded result is a denormal, otherwise cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if either of the input operands is a NAN, or if they are opposite-signed infinities, otherwise cleared

Fn = Fx – Fy

Function

Subtracts the floating-point operand in register Fy from the floating-point operand in register Fx. The normalized result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in `MODE1`. Post-rounded overflow returns ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±zero. Denormal inputs are flushed to ±zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Set if the post-rounded result is a denormal, otherwise cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if either of the input operands is a NAN, or if they are like-signed infinities, otherwise cleared

Fn = ABS (Fx + Fy)

Function

Adds the floating-point operands in registers Fx and Fy, and places the absolute value of the normalized result in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in `MODE1`.

Post-rounded overflow returns +infinity (round-to-nearest) or +NORM.MAX (round-to-zero). Post-rounded denormal returns +zero. Denormal inputs are flushed to ±zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Set if the post-rounded result is a denormal, otherwise cleared
AN	Cleared
AV	Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if either of the input operands is a NAN, or if they are opposite-signed infinities, otherwise cleared

Fn = ABS (Fx – Fy)

Function

Subtracts the floating-point operand in Fy from the floating-point operand in Fx and places the absolute value of the normalized result in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in `MODE1`. Post-rounded overflow returns +infinity (round-to-nearest) or +NORM.MAX (round-to-zero). Post-rounded denormal returns +zero. Denormal inputs are flushed to ±zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Set if the post-rounded result is a denormal, otherwise cleared
AN	Cleared
AV	Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if either of the input operands is a NAN, or if they are like-signed infinities, otherwise cleared

Fn = (Fx + Fy)/2

Function

Adds the floating-point operands in registers Fx and Fy and divides the result by 2, by decrementing the exponent of the sum before rounding. The normalized result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in `MODE1`. Post-rounded overflow returns ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal results return ±zero. A denormal input is flushed to ±zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Set if the post-rounded result is a denormal, otherwise cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if either of the input operands is a NAN, or if they are opposite-signed infinities, otherwise cleared

COMP(Fx, Fy)

Function

Compares the floating-point operand in register Fx with the floating-point operand in register Fy. Sets the AZ flag if the two operands are equal, and the AN flag if the operand in register Fx is smaller than the operand in register Fy.

The ASTAT register stores the results of the previous eight ALU compare operations in bits 24-31. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed. The MSB of ASTAT is set if the X operand is greater than the Y operand (its value is the AND of $\overline{\text{AZ}}$ and $\overline{\text{AN}}$); it is otherwise cleared.

Status Flags

AZ	Set if the operands in registers Fx and Fy are equal, otherwise cleared
AU	Cleared
AN	Set if the operand in the Fx register is smaller than the operand in the Fy register, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Set if either of the input operands is a NAN, otherwise cleared

Fn = –Fx

Function

Complements the sign bit of the floating-point operand in Fx. The complemented result is placed in register Fn. A denormal input is flushed to ±zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the result operand is a ±zero, otherwise cleared
AU	Cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Set if the input operand is a NAN, otherwise cleared

Fn = ABS Fx

Function

Returns the absolute value of the floating-point operand in register Fx by setting the sign bit of the operand to 0. Denormal inputs are flushed to +zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the result operand is +zero, otherwise cleared
AU	Cleared
AN	Cleared
AV	Cleared
AC	Cleared
AS	Set if the input operand is negative, otherwise cleared
AI	Set if the input operand is a NAN, otherwise cleared

Fn = PASS Fx

Function

Passes the floating-point operand in Fx through the ALU to the floating-point field in register Fn. Denormal inputs are flushed to ±zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the result operand is a ±zero, otherwise cleared
AU	Cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Set if the input operand is a NAN, otherwise cleared

Fn = RND Fx

Function

Rounds the floating-point operand in register Fx to a 32 bit boundary. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in `MODE1`. Post-rounded overflow returns ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). A denormal input is flushed to ±zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the result operand is a ±zero, otherwise cleared
AU	Cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if the input operand is a NAN, otherwise cleared

Fn = SCALB Fx BY Ry

Function

Scales the exponent of the floating-point operand in Fx by adding to it the fixed-point two's-complement integer in Ry. The scaled floating-point result is placed in register Fn. Overflow returns ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Denormal returns ±zero. Denormal inputs are flushed to ±zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Set if the post-rounded result is a denormal, otherwise cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Set if the result overflows (unbiased exponent > +127), otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if the input is a NAN, an otherwise cleared

Rn = MANT Fx

Function

Extracts the mantissa (fraction bits with explicit hidden bit, excluding the sign bit) from the floating-point operand in Fx. The unsigned-magnitude result is left-justified (1.31 format) in the fixed-point field in Rn. Rounding modes are ignored and no rounding is performed because all results are inherently exact. Denormal inputs are flushed to ±zero. A NAN or an infinity input returns an all 1s result (–1 in signed fixed-point format).

Status Flags

AZ	Set if the result is zero, otherwise cleared
AU	Cleared
AN	Cleared
AV	Cleared
AC	Cleared
AS	Set if the input is negative, otherwise cleared
AI	Set if the input operands is a NAN or an infinity, otherwise cleared

Rn = LOGB Fx

Function

Converts the exponent of the floating-point operand in register Fx to an unbiased two's-complement fixed-point integer. The result is placed in the fixed-point field in register Rn. Unbiasing is done by subtracting 127 from the floating-point exponent in Fx. If saturation mode is not set, a ±infinity input returns a floating-point +infinity and a ±zero input returns a floating-point –infinity. If saturation mode is set, a ±infinity input returns the maximum positive value (0x7FFF FFFF), and a ±zero input returns the maximum negative value (0x8000 0000). Denormal inputs are flushed to ±zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the fixed-point result is zero, otherwise cleared
AU	Cleared
AN	Set if the result is negative, otherwise cleared
AV	Set if the input operand is an infinity or a zero, otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if the input is a NAN, otherwise cleared

Rn = FIX Fx
Rn = TRUNC Fx
Rn = FIX Fx BY Ry
Rn = TRUNC Fx BY Ry

Function

Converts the floating-point operand in Fx to a two's-complement 32-bit fixed-point integer result.

If the `MODE1` register `TRUNC` bit=1, the Fix operation truncates the mantissa towards –infinity. If the `TRUNC` bit=0, the Fix operation rounds the mantissa towards the nearest integer.

The trunc operation always truncates toward 0. The `TRUNC` bit does not influence operation of the trunc instruction.

If a scaling factor (Ry) is specified, the fixed-point two's-complement integer in Ry is added to the exponent of the floating-point operand in Fx before the conversion.

The result of the conversion is right-justified (32.0 format) in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

In saturation mode (the ALU saturation mode bit in `MODE1` set) positive overflows and +infinity return the maximum positive number (0x7FFF FFFF), and negative overflows and –infinity return the minimum negative number (0x8000 0000).

For the Fix operation, rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in `MODE1`. A NAN input returns a floating-point all 1s result. If saturation mode is not set, an infinity input or a result that overflows returns a floating-point result of all 1s.

All positive underflows return zero. Negative underflows that are rounded-to-nearest return zero, and negative underflows that are rounded by truncation return –1 (0xFF FFFF FF00).

Status Flags

AZ	Set if the fixed-point result is zero, otherwise cleared
AU	Set if the pre-rounded result is a denormal, otherwise cleared
AN	Set if the fixed-point result is negative, otherwise cleared
AV	Set if the conversion causes the floating-point mantissa to be shifted left, that is, if the floating-point exponent + scale bias is >157 (127 + 31 – 1) or if the input is ±infinity, otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if the input operand is a NAN or, when saturation mode is not set, either input is an infinity or the result overflows, otherwise cleared

Fn = FLOAT Rx BY Ry
Fn = FLOAT Rx

Function

Converts the fixed-point operand in Rx to a floating-point result. If a scaling factor (Ry) is specified, the fixed-point two's-complement integer in Ry is added to the exponent of the floating-point result. The final result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode, to a 40-bit boundary, regardless of the values of the rounding boundary bits in `MODE1`. The exponent scale bias may cause a floating-point overflow or a floating-point underflow. Overflow generates a return of ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero); underflow generates a return of ±zero.

Status Flags

AZ	Set if the result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Set if the post-rounded result is a denormal, otherwise cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Set if the result overflows (unbiased exponent >127)
AC	Cleared
AS	Cleared
AI	Cleared

Fn = RECIPS Fx

Function

Creates an 8-bit accurate seed for 1/Fx, the reciprocal of Fx. The mantissa of the seed is determined from a ROM table using the 7 MSBs (excluding the hidden bit) of the Fx mantissa as an index. The unbiased exponent of the seed is calculated as the two's-complement of the unbiased Fx exponent, decremented by one; that is, if e is the unbiased exponent of Fx, then the unbiased exponent of Fn = –e – 1. The sign of the seed is the sign of the input. A ±zero returns ±infinity and sets the overflow flag. If the unbiased exponent of Fx is greater than +125, the result is ±zero. A NAN input returns an all 1s result.

The following code performs floating-point division using an iterative convergence algorithm.[1] The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set. The following inputs are required: F0=numerator, F12=denominator, F11=2.0. The quotient is returned in F0. (The two highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

```
F0=RECIPS F12, F7=F0;   {Get 8 bit seed R0=1/D}
F12=F0*F12;   {D' = D*R0}
F7=F0*F7, F0=F11-F12;   {F0=R1=2-D', F7=N*R0}
F12=F0*F12;   {F12=D'-D'*R1}
F7=F0*F7, F0=F11-F12;   {F7=N*R0*R1, F0=R2=2-D'}
F12=F0*F12;   {F12=D'=D'*R2}
F7=F0*F7, F0=F11-F12;   {F7=N*R0*R1*R2, F0=R3=2-D'}
F0=F0*F7;   {F7=N*R0*R1*R2*R3}
```

To make this code segment a subroutine, add an RTS(DB) clause to the third-to-last instruction.

[1] Cavanagh, J. 1984. Digital Computer Arithmetic. McGraw-Hill. Page 284.

Status Flags

AZ	Set if the floating-point result is ±zero (unbiased exponent of Fx is greater than +125), otherwise cleared
AU	Cleared
AN	Set if the input operand is negative, otherwise cleared
AV	Set if the input operand is ±zero, otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if the input operand is a NAN, otherwise cleared

Fn = RSQRTS Fx

Function

Creates a 4-bit accurate seed for $1/(Fx)^{½}$, the reciprocal square root of Fx.

The mantissa of the seed is determined from a ROM table, using the LSB of the biased exponent of Fx concatenated with the six MSBs (excluding the hidden bit of the mantissa) of Fx's index.

The unbiased exponent of the seed is calculated as the two's-complement of the unbiased Fx exponent, shifted right by one bit and decremented by one; that is, if e is the unbiased exponent of Fx, then the unbiased exponent of Fn = –INT[e/2] – 1.

The sign of the seed is the sign of the input. The input ±zero returns ±infinity and sets the overflow flag. The input +infinity returns +zero. A NAN input or a negative nonzero input returns a result of all 1s.

The following code calculates a floating-point reciprocal square root ($1/(x)^{½}$) using a Newton-Raphson iteration algorithm.[1] The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set.

To calculate the square root, simply multiply the result by the original input. The following inputs are required: F0=input, F8=3.0, F1=0.5. The result is returned in F4. (The four highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

```
F4=RSQRTS F0;    {Fetch 4-bit seed}
F12=F4*F4;       {F12=X0^2}
F12=F12*F0;      {F12=C*X0^2}
F4=F1*F4, F12=F8-F12;   {F4=.5*X0, F12=3-C*X0^2}
F4=F4*F12;       {F4=X1=.5*X0(3-C*X0^2)}
F12=F4*F4;       {F12=X1^2}
F12=F12*F0;   {F12=C*X1^2}
```

[1] Cavanagh, J. 1984. Digital Computer Arithmetic. McGraw-Hill. Page 278.

```
F4=F1*F4, F12=F8-F12;   {F4=.5*X1, F12=3-C*X1^2}
F4=F4*F12;    {F4=X2=.5*X1(3-C*X1^2)}
F12=F4*F4;    {F12=X2^2}
F12=F12*F0;   {F12=C*X2^2}
F4=F1*F4, F12=F8-F12;   {F4=.5*X2, F12=3-C*X2^2}
F4=F4*F12;    {F4=X3=.5*X2(3-C*X2^2)}
```

Note that this code segment can be made into a subroutine by adding an RTS(DB) clause to the third-to-last instruction.

Status Flags

AZ	Set if the floating-point result is +zero (Fx = +infinity), otherwise cleared
AU	Cleared
AN	Set if the input operand is –zero, otherwise cleared
AV	Set if the input operand is ±zero, otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if the input operand is negative and nonzero, or a NAN, otherwise cleared

Fn = Fx COPYSIGN Fy

Function

Copies the sign of the floating-point operand in register Fy to the floating-point operand from register Fx without changing the exponent or the mantissa. The result is placed in register Fn. A denormal input is flushed to ±zero. A NAN input returns an all 1s result.

Status Flags

AZ	Set if the floating-point result is ±zero, otherwise cleared
AU	Cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Set if either of the input operands is a NAN, otherwise cleared

Fn = MIN(Fx, Fy)

Function

Returns the smaller of the floating-point operands in register Fx and Fy. A NAN input returns an all 1s result. The MIN of +zero and –zero returns –zero. Denormal inputs are flushed to ±zero.

Status Flags

AZ	Set if the floating-point result is ±zero, otherwise cleared
AU	Cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Set if either of the input operands is a NAN, otherwise cleared

Fn = MAX(Fx, Fy)

Function

Returns the larger of the floating-point operands in registers Fx and Fy. A NAN input returns an all 1s result. The MAX of +zero and –zero returns +zero. Denormal inputs are flushed to ±zero.

Status Flags

AZ	Set if the floating-point result is ±zero, otherwise cleared
AU	Cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Set if either of the input operands is a NAN, otherwise cleared

Fn = CLIP Fx BY Fy

Function

Returns the floating-point operand in Fx if the absolute value of the operand in Fx is less than the absolute value of the floating-point operand in Fy. Else, returns | Fy | if Fx is positive, and –| Fy | if Fx is negative. A NAN input returns an all 1s result. Denormal inputs are flushed to ±zero.

Status Flags

AZ	Set if the floating-point result is ±zero, otherwise cleared
AU	Cleared
AN	Set if the floating-point result is negative, otherwise cleared
AV	Cleared
AC	Cleared
AS	Cleared
AI	Set if either of the input operands is a NAN, otherwise cleared

Multiplier Operations

This section describes the multiplier operations. These tables use the following symbols to indicate location of operands and other features:

- y = y-input (1 = signed, 0 = unsigned)
- x = x-input (1 = signed, 0 = unsigned)
- f = format (1 = fractional, 0 = integer)
- r = rounding (1 = yes, 0 = no)

Multiplier Fixed-Point Operations

Table 9-3 summarizes the syntax and opcodes for the fixed-point multiplier operations.

Table 9-3. Multiplier Fixed-Point Operations

Syntax	Opcode	Reference Page
Rn = Rx * Ry *mod2*	01yx f00r	on page 9-54
MRF = Rx * Ry *mod2*	01yx f10r	on page 9-54
MRB = Rx * Ry *mod2*	01yx f11r	on page 9-54
Rn = MRF + Rx * Ry *mod2*	10yx f00r	on page 9-55
Rn = MRB + Rx * Ry *mod2*	10yx f01r	on page 9-55
MRF = MRF + Rx * Ry *mod2*	10yx f10r	on page 9-55
MRB = MRB + Rx * Ry *mod2*	10yx f11r	on page 9-55
Rn = MRF – Rx * Ry *mod2*	11yx f00r	on page 9-56
Rn = MRB – Rx * Ry *mod2*	11yx f01r	on page 9-56
MRF = MRF – Rx * Ry *mod2*	11yx f10r	on page 9-56
MRB = MRB – Rx * Ry *mod2*	11yx f11r	on page 9-56
Rn = SAT MRF *mod1*	0000 f00x	on page 9-57
Rn = SAT MRB *mod1*	0000 f01x	on page 9-57
MRF = SAT MRF *mod1*	0000 f10x	on page 9-57
MRB = SAT MRB *mod1*	0000 f11x	on page 9-57
Rn = RND MRF *mod1*	0001 100x	on page 9-58
Rn = RND MRB *mod1*	0001 101x	on page 9-58
MRF = RND MRF *mod1*	0001 110x	on page 9-58
MRB = RND MRB *mod1*	0001 111x	on page 9-58
MRF = 0	0001 0100	on page 9-59
MRB = 0	0001 0110r	on page 9-59

Table 9-3. Multiplier Fixed-Point Operations (Cont'd)

Syntax	Opcode	Reference Page
MR = Rn		on page 9-60
Rn = MR		on page 9-60

Multiplier Floating-Point Operations

Table 9-4 summarizes the syntax and opcodes for the floating-point multiplier operations.

Table 9-4. Multiplier Floating-Point Operations

Syntax	Opcode	Reference Page
Fn = Fx*Fy	0011 0000	on page 9-62

Mod1 and Mod2 Modifiers

Mod2 in Table 9-3 on page 9-51 is an optional modifier. It is enclosed in parentheses and consists of three or four letters that indicate whether:

- The x-input is signed (S) or unsigned (U).
- The y-input is signed or unsigned.
- The inputs are in integer (I) or fractional (F) format.
- The result written to the register file will be rounded-to-nearest (R).

Table 9-5 lists the options for mod2 and the corresponding opcode values.

Table 9-5. Mod2 Options and Opcodes

Option	Opcode
(SSI)	_ _11 0_ _0
(SUI)	_ _01 0_ _0
(USI)	_ _10 0_ _0
(UUI)	_ _00 0_ _0
(SSF)	_ _11 1_ _0
(SUF)	_ _01 1_ _0
(USF)	_ _10 1_ _0
(UUF)	_ _00 1_ _0
(SSFR)	_ _11 1_ _1
(SUFR)	_ _01 1_ _1
(USFR)	_ _10 1_ _1
(UUFR)	_ _00 1_ _1

Similarly, mod1 in Table 9-3 on page 9-51 is an optional modifier, enclosed in parentheses, consisting of two letters that indicate whether the input is signed (S) or unsigned (U) and whether the input is in integer (I) or fractional (F) format. The options for mod1 and the corresponding opcode values are listed in Table 9-6.

Table 9-6. Mod1 Options and Opcodes

Option	Opcode
(SI) (for SAT only)	_ _ _ _ 0 _ _ 1
(UI) (for SAT only)	_ _ _ _ 0 _ _ 0
(SF)	_ _ _ _ 1 _ _ 1
(UF)	_ _ _ _ 1 _ _ 0

Rn = Rx * Ry mod2
MRF = Rx * Ry mod2
MRB Rx * Ry mod2

Function

Multiplies the fixed-point fields in registers Rx and Ry.

If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register Rn or one of the `MR` accumulation registers.

If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31–0 for integers, bits 63–32 for fractional). The floating-point extension field in Rn is set to all 0s. If `MRF` or `MRB` is specified, the entire 80-bit result is placed in `MRF` or `MRB`.

Status Flags

MN	Set if the result is negative, otherwise cleared
MV	Set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result); number of upper bits depends on format; for a signed result, fractional=33, integer=49; for an unsigned result, fractional=32, integer=48
MU	Set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros; integer results do not underflow
MI	Cleared

Rn = MRF + Rx * Ry mod2
Rn = MRB + Rx * Ry mod2
MRF = MRF + Rx * Ry mod2
MRB = MRB + Rx * Ry mod2

Function

Multiplies the fixed-point fields in registers Rx and Ry, and adds the product to the specified `MR` register value. If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register Rn or one of the `MR` accumulation registers, which must be the same `MR` register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31–0 for integers, bits 63–32 for fractional). The floating-point extension field in Rn is set to all 0s. If `MRF` or `MRB` is specified, the entire 80-bit result is placed in `MRF` or `MRB`.

Status Flags

MN	Set if the result is negative, otherwise cleared
MV	Set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result); number of upper bits depends on format; for a signed result, fractional=33, integer=49; for an unsigned result, fractional=32, integer=48
MU	Set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros; integer results do not underflow
MI	Cleared

Rn = MRF – Rx * Ry mod2
Rn = MRB – Rx * Ry mod2
MRF = MRF – Rx * Ry mod2
MRB = MRB – Rx * Ry mod2

Function

Multiplies the fixed-point fields in registers Rx and Ry, and subtracts the product from the specified MR register value. If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register Rn or in one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31–0 for integers, bits 63–32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status Flags

MN	Set if the result is negative, otherwise cleared
MV	Set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result); number of upper bits depends on format; for a signed result, fractional=33, integer=49; for an unsigned result, fractional=32, integer=48
MU	Set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros; integer results do not underflow
MI	Cleared

Rn = SAT MRF mod1
Rn = SAT MRB mod1
MRF = SAT MRF mod1
MRB = SAT MRB mod1

Function

If the value of the specified `MR` register is greater than the maximum value for the specified data format, the multiplier sets the result to the maximum value. Otherwise, the `MR` value is unaffected. The result is placed either in the fixed-point field in register Rn or one of the `MR` accumulation registers, which must be the same `MR` register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31–0 for integers, bits 63–32 for fractional). The floating-point extension field in Rn is set to all 0s. If `MRF` or `MRB` is specified, the entire 80-bit result is placed in `MRF` or `MRB`.

Status Flags

MN	Set if the result is negative, otherwise cleared
MV	Cleared
MU	Set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros; integer results do not underflow
MI	Cleared

Rn = RND MRF mod1
Rn = RND MRB mod1
MRF = RND MRF mod1
MRB = RND MRB mod1

Function

Rounds the specified MR value to nearest at bit 32 (the MR1–MR0 boundary). The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31–0 for integers, bits 63–32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status Flags

MN	Set if the result is negative, otherwise cleared
MV	Set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result); number of upper bits depends on format; for a signed result, fractional=33, integer=49; for an unsigned result, fractional=32, integer=48
MU	Set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros; integer results do not underflow
MI	Cleared

MRF = 0
MRB = 0

Function

Sets the value of the specified `MR` register to zero. All 80 bits (`MR2`, `MR1`, `MR0`) are cleared.

Status Flags

MN	Not Affected
MV	Not Affected
MU	Not Affected
MI	Not Affected

MRxF/B = Rn/Rn = MRxF/B

Function

A transfer to an MR register places the fixed-point field of register Rn in the specified MR register. The floating-point extension field in Rn is ignored. A transfer from an MR register places the specified MR register in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Syntax Variations

```
MR0F = Rn    Rn = MR0F
MR1F = Rn    Rn = MR1F
MR2F = Rn    Rn = MR2F
MR0B = Rn    Rn = MR0B
MR1B = Rn    Rn = MR1B
MR2B = Rn    Rn = MR2B
```

Compute Field

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
100000						T	Ai				Rk											

Table 9-7 indicates how Ai specifies the MR register, and Rk specifies the data register. The T determines the direction of the transfer (0=to register file, 1=to MR register).

Table 9-7. Ai Values and MR Registers

Ai	MR Register
0000	MR0F
0001	MR1F

Table 9-7. Ai Values and MR Registers (Cont'd)

Ai	MR Register
0010	MR2F
0100	MR0B
0101	MR1B
0110	MR2B

Status Flags

MN	Not Affected
MV	Not Affected
MU	Not Affected
MI	Not Affected

Fn = Fx * Fy

Function

Multiplies the floating-point operands in registers Fx and Fy and places the result in the register Fn.

Status Flags

MN	Set if the result is negative, otherwise cleared
MV	Set if the unbiased exponent of the result is greater than 127, otherwise cleared
MU	Set if the unbiased exponent of the result is less than –126, otherwise cleared
MI	Set if either input is a NAN or if the inputs are ±infinity and ±zero, otherwise cleared

Shifter Operations

Shifter operations are described in this section. Table 9-8 lists the syntax and opcodes for the shifter operations. The succeeding pages provide detailed descriptions of each operation. Some of the instructions in Table 9-8 accept the following modifiers.

- (SE) = Sign extension of deposited or extracted field
- (EX) = Extended exponent extract

Shifter Opcodes

The shifter operates on the register file's 32-bit fixed-point fields (bits 38–9). Two-input shifter operations can take their y input from the register file or from immediate data provided in the instruction. Either form uses the same opcode. However, the latter case, called an immediate shift or shifter immediate operation, is allowed only with instruction type 6, which has an immediate data field in its opcode for this purpose. All other instruction types must obtain the y input from the register file when the compute operation is a two-input shifter operation.

Table 9-8. Shifter Operations

Syntax	Opcode	Reference Page
Rn = LSHIFT Rx BY Ry|<data8>	0000 0000	on page 9-64
Rn = Rn OR LSHIFT Rx BY Ry|<data8>	0010 0000	on page 9-65
Rn = ASHIFT Rx BY Ry|<data8>	0000 0100	on page 9-66
Rn = Rn OR ASHIFT Rx BY Ry|<data8>	0010 0100	on page 9-67
Rn = ROT Rx BY Ry|<data8>	0000 1000	on page 9-68
Rn = BCLR Rx BY Ry|<data8>	1100 0100	on page 9-69
Rn =BSET Rx BY Ry|<data8>	1100 0000	on page 9-70
Rn = BTGL Rx BY Ry|<data8>	1100 1000	on page 9-71
BTST Rx BY Ry|<data8>	1100 1100	on page 9-72
Rn = FDEP Rx BY Ry|<bit6>:<len6>	0100 0100	on page 9-73
Rn = Rn OR FDEP Rx BY Ry|<bit6>:<len6>	0110 0100	on page 9-75
Rn = FDEP Rx BY Ry|<bit6>:<len6> (SE)	0100 1100	on page 9-77
Rn = Rn OR FDEP Rx BY Ry|<bit6>:<len6>(SE)	0110 1100	on page 9-79
Rn = FEXT RX BY Ry|<bit6>:<len6>	0100 0000	on page 9-81
Rn = FEXT Rx BY Ry|<bit6>:<len6> (SE)	0100 1000	on page 9-83
Rn = EXP Rx	1000 0000	on page 9-85
Rn = EXP Rx (EX)	1000 0100	on page 9-86
Rn = LEFTZ Rx	1000 1000	on page 9-87
Rn = LEFTO Rx	1000 1100	on page 9-88
Rn = FPACK Fx	1001 0000	on page 9-89
Fn = FUNPACK Rx	1001 0100	on page 9-90

Rn = LSHIFT Rx BY Ry
Rn = LSHIFT Rx BY <data8>

Function

Logically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are two's-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status Flags

SZ	Set if the shifted result is zero, otherwise cleared
SV	Set if the input is shifted to the left by more than 0, otherwise cleared
SS	Cleared

Rn = Rn OR LSHIFT Rx BY Ry
Rn = Rn OR LSHIFT Rx BY <data8>

Function

Logically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is logically ORed with the fixed-point field of register Rn and then written back to register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are two's-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status Flags

SZ	Set if the shifted result is zero, otherwise cleared
SV	Set if the input is shifted left by more than 0, otherwise cleared
SS	Cleared

Rn = ASHIFT Rx BY Ry
Rn = ASHIFT Rx BY <data8>

Function

Arithmetically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are two's-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status Flags

SZ	Set if the shifted result is zero, otherwise cleared
SV	Set if the input is shifted left by more than 0, otherwise cleared
SS	Cleared

Rn = Rn OR ASHIFT Rx BY Ry
Rn = Rn OR ASHIFT Rx BY <data8>

Function

Arithmetically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is logically ORed with the fixed-point field of register Rn and then written back to register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are two's-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status Flags

SZ	Set if the shifted result is zero, otherwise cleared
SV	Set if the input is shifted left by more than 0, otherwise cleared
SS	Cleared

Rn = ROT Rx BY Ry
Rn = ROT Rx BY <data8>

Function

Rotates the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The rotated result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are two's-complement numbers. Positive values select a rotate left; negative values select a rotate right. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a rotate of a 32-bit field from full right wrap around to full left wrap around.

Status Flags

SZ	Set if the rotated result is zero, otherwise cleared
SV	Cleared
SS	Cleared

Rn = BCLR Rx BY Ry
Rn = BCLR Rx BY <data8>

Function

Clears a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be cleared. If the bit position value is greater than 31 or less than 0, no bits are cleared.

Status Flags

SZ	Set if the output operand is 0, otherwise cleared
SV	Set if the bit position is greater than 31, otherwise cleared
SS	Cleared

This compute operation affects a bit in a register file location. There is also a bit manipulation instruction that affects one or more bits in a system register. The Bit Clr instruction should not be confused with the Bclr shifter operation. For more information on Bit Clr, see "Type 18: System Register Bit Manipulation" on page 8-72.

Rn = BSET Rx BY Ry
Rn = BSET Rx BY <data8>

Function

Sets a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be set. If the bit position value is greater than 31 or less than 0, no bits are set.

Status Flags

SZ	Set if the output operand is 0, otherwise cleared
SV	Set if the bit position is greater than 31, otherwise cleared
SS	Cleared

This compute operation affects a bit in a register file location. There is also a bit manipulation instruction that affects one or more bits in a system register. This Bit Set instruction should not be confused with the Bset shifter operation. For more information on Bit Set, see "Type 18: System Register Bit Manipulation" on page 8-72.

Rn = BTGL Rx BY Ry
Rn = BTGL Rx BY <data8>

Function

Toggles a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be toggled. If the bit position value is greater than 31 or less than 0, no bits are toggled.

Status Flags

SZ	Set if the output operand is 0, otherwise cleared
SV	Set if the bit position is greater than 31, otherwise cleared
SS	Cleared

ⓘ This compute operation affects a bit in a register file location. There is also a bit manipulation instruction that affects one or more bits in a system register. This Bit Tgl instruction should not be confused with the Btgl shifter operation. For more information on Bit Tgl, see "Type 18: System Register Bit Manipulation" on page 8-72.

BTST Rx BY Ry
BTST Rx BY <data8>

Function

Tests a bit in the fixed-point operand in register Rx. The SZ flag is set if the bit is a 0 and cleared if the bit is a 1. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be tested. If the bit position value is greater than 31 or less than 0, no bits are tested.

Status Flags

SZ	Cleared if the tested bit is a 1, is set if the tested bit is a 0 or if the bit position is greater than 31
SV	Set if the bit position is greater than 31, otherwise cleared
SS	Cleared

This compute operation tests a bit in a register file location. There is also a bit manipulation instruction that tests one or more bits in a system register. This Bit Tst instruction should not be confused with the Btst shifter operation.

For more information on Bit Tst, see "Type 18: System Register Bit Manipulation" on page 8-72.

Rn = FDEP Rx BY Ry
Rn = FDEP Rx BY <bit6>:<len6>

Function

Deposits a field from register Rx to register Rn. (See Figure 9-1.) The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bits to the left and to the right of the deposited field are set to 0. The floating-point extension field of Rn (bits 7–0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits, and to bit positions ranging from 0 to off-scale left.

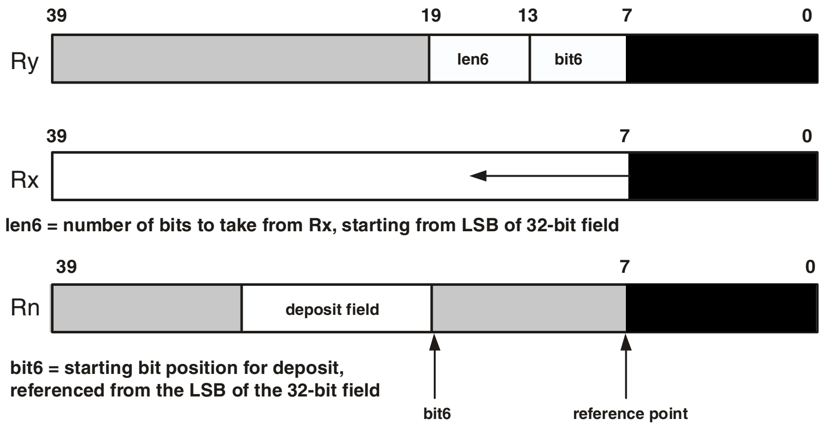

39
19
13
7
0
Ry
len6
bit6
39
7
0
Rx
len6 = number of bits to take from Rx, starting from LSB of 32-bit field
39
7
0
Rn
deposit field
bit6 = starting bit position for deposit, referenced from the LSB of the 32-bit field
bit6
reference point

Figure 9-1. Field Alignment

Example

If len6=14 and bit6=13, then the 14 bits of Rx are deposited in Rn bits 34–21 (of the 40-bit word).

```
39       31       23       15       7        0
|--------|--------|--abcdef|ghijklmn|--------|           Rx
                     \-------------/
                            14 bits

39       31       23       15       7        0
|00000abc|defghijk|lmn00000|00000000|00000000|           Rn
      \--------------/
                      |
                       bit position 13 (from reference point)
```

Status Flags

SZ	Set if the output operand is 0, otherwise cleared
SV	Set if any bits are deposited to the left of the 32-bit fixed-point output field (that is, if len6 + bit6 > 32), otherwise cleared
SS	Cleared

Rn = Rn OR FDEP Rx BY Ry
Rn = Rn OR FDEP Rx BY <bit6>:<len6>

Function

Deposits a field from register Rx to register Rn. The field value is logically ORed bitwise with the specified field of register Rn and the new value is written back to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction.

The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits, and to bit positions ranging from 0 to off-scale left.

Example

```
39       31       23       15       7        0
|--------|--------|--abcdef|ghijklmn|--------|      Rx
                     \--------------/
                           len6 bits

39       31       23       15       7        0
|abcdefgh|ijklmnop|qrstuvwx|yzabcdef|ghijklmn|      Rn old
      \--------------/
                     |
                     bit position bit6 (from reference point)

39       31       23       15       7        0

|abcdeopq|rstuvwxy|zabtuvwx|yzabcdef|ghijklmn|      Rn new
      └──────────────┘
              OR result
```

Status Flags

SZ	Set if the output operand is 0, otherwise cleared
SV	Set if any bits are deposited to the left of the 32-bit fixed-point output field (that is, if len6 + bit6 > 32), otherwise cleared
SS	Cleared

Rn = FDEP Rx BY Ry (SE)
Rn = FDEP Rx BY <bit6>:<len6> (SE)

Function

Deposits and sign-extends a field from register Rx to register Rn. (See Figure 9-2.) The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. The MSBs of Rn are sign-extended by the MSB of the deposited field, unless the MSB of the deposited field is off-scale left. Bits to the right of the deposited field are set to 0. The floating-point extension field of Rn (bits 7–0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits into bit positions ranging from 0 to off-scale left.

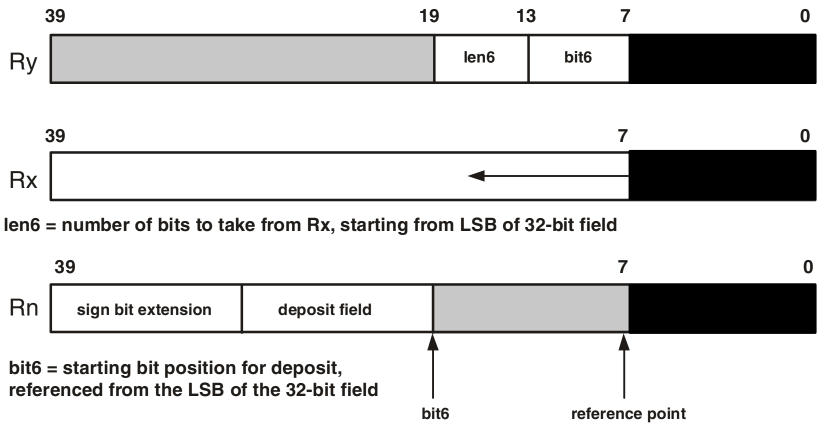

39
19
13
7
0
Ry
len6
bit6
39
7
0
Rx
len6 = number of bits to take from Rx, starting from LSB of 32-bit field
39
7
0
Rn
sign bit extension
deposit field
bit6 = starting bit position for deposit,
referenced from the LSB of the 32-bit field
bit6
reference point

Figure 9-2. Field Alignment

Example

```
39       31       23       15       7        0
|--------|--------|--abcdef|ghijklmn|--------|          Rx
                    \---------------/
                          len6 bits

39       31       23       15       7        0
|aaaaaabc|defghijk|lmn00000|00000000|00000000|          Rn
\----/\--------------/
 sign                |
 extension           bit position bit6
                     (from reference point)
```

Status Flags

SZ	Set if the output operand is 0, otherwise cleared
SV	Set if any bits are deposited to the left of the 32-bit fixed-point output field (that is, if len6 + bit6 > 32), otherwise cleared
SS	Cleared

Rn = Rn OR FDEP Rx BY Ry (SE)
Rn = Rn OR FDEP Rx BY <bit6>:<len6> (SE)

Function

Deposits and sign-extends a field from register Rx to register Rn. The sign-extended field value is logically ORed bitwise with the value of register Rn and the new value is written back to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry.

The bit position can also be determined by the immediate bit6 field in the instruction. Bit6 and len6 can take values between 0 and 63 inclusive to allow the deposit of fields ranging in length from 0 to 32 bits into bit positions ranging from 0 to off-scale left.

Example

```
39       31       23       15       7        0
|--------|--------|--abcdef|ghijklmn|--------|          Rx
                    \-------------/
                          len6 bits

39       31       23       15       7        0
|aaaaaabc|defghijk|lmn00000|00000000|00000000|
\----/\--------------/
  sign               |
extension             bit position bit6
                     (from reference point)

39       31       23       15       7        0
|abcdefgh|ijklmnop|qrstuvwx|yzabcdef|ghijklmn|      Rn old
```

```
 39       31       23       15       7        0

|vwxyzabc|defghijk|lmntuvwx|yzabcdef|ghijklmn|      Rn new
          |
           OR result
```

Status Flags

SZ	Set if the output operand is 0, otherwise cleared
SV	Set if any bits are deposited to the left of the 32-bit fixed-point output field (that is, if len6 + bit6 > 32), otherwise cleared
SS	Cleared

Rn = FEXT Rx BY Ry
Rn = FEXT Rx BY <bit6>:<len6>

Function

Extracts a field from register Rx to register Rn. (See Figure 9-3.) The output field is placed right-aligned in the fixed-point field of Rn. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is extracted from the fixed-point field of Rx starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bits to the left of the extracted field are set to 0 in register Rn. The floating-point extension field of Rn (bits 7–0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for extraction of fields ranging in length from 0 to 32 bits, and from bit positions ranging from 0 to off-scale left.

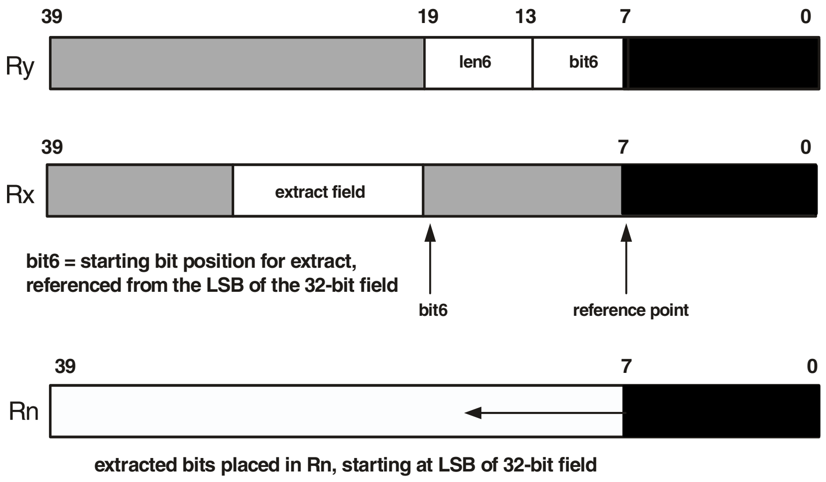

39
19
13
7
0
Ry
len6
bit6
39
7
0
Rx
extract field
bit6 = starting bit position for extract, referenced from the LSB of the 32-bit field
bit6
reference point
39
7
0
Rn
extracted bits placed in Rn, starting at LSB of 32-bit field

Figure 9-3. Field Alignment

Example

```
39       31       23       15       7        0
|-----abc|defghijk|lmn-----|--------|--------|            Rx
      \--------------/
      len6 bits     |
                    bit position bit6
                    (from reference point)

39       31       23       15       7        0
|00000000|00000000|00abcdef|ghijklmn|00000000|           Rn
```

Status Flags

SZ	Set if the output operand is 0, otherwise cleared
SV	Set if any bits are extracted from the left of the 32-bit fixed-point, input field (that is, if len6 + bit6 > 32), otherwise cleared
SS	Cleared

Rn = FEXT Rx BY Ry (SE)
Rn = FEXT Rx BY <bit6>:<len6> (SE)

Function

Extracts and sign-extends a field from register Rx to register Rn. The output field is placed right-aligned in the fixed-point field of Rn. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is extracted from the fixed-point field of Rx starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. The MSBs of Rn are sign-extended by the MSB of the extracted field, unless the MSB is extracted from off-scale left.

The floating-point extension field of Rn (bits 7–0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for extraction of fields ranging in length from 0 to 32 bits and from bit positions ranging from 0 to off-scale left.

Example

```
39       31       23       15       7        0
|-----abc|defghijk|lmn-----|--------|--------|          Rx
      \--------------/
         len6 bits   |
                     bit position bit6
                     (from reference point)

39       31       23       15       7        0
|aaaaaaaa|aaaaaaaa|aaabcdef|ghijklmn|00000000|          Rn
\-------------------/
    sign extension
```

Status Flags

SZ	Set if the output operand is 0, otherwise cleared
SV	Set if any bits are extracted from the left of the 32-bit fixed-point input field (that is, if len6 + bit6 > 32), otherwise cleared
SS	Cleared

Rn = EXP Rx

Function

Extracts the exponent of the fixed-point operand in Rx. The exponent is placed in the shf8 field in register Rn. The exponent is calculated as the two's-complement of:

```
# leading sign bits in Rx - 1
```

Status Flags

SZ	Set if the extracted exponent is 0, otherwise cleared
SV	Cleared
SS	Set if the fixed-point operand in Rx is negative (bit 31 is a 1), otherwise cleared

Rn = EXP Rx (EX)

Function

Extracts the exponent of the fixed-point operand in Rx, assuming that the operand is the result of an ALU operation. The exponent is placed in the shf8 field in register Rn. If the AV status bit is set, a value of +1 is placed in the shf8 field to indicate an extra bit (the ALU overflow bit). If the AV status bit is not set, the exponent is calculated as the two's-complement of:

```
# leading sign bits in Rx – 1
```

Status Flags

SZ	Set if the extracted exponent is 0, otherwise cleared
SV	Cleared
SS	Set if the exclusive OR of the AV status bit and the sign bit (bit 31) of the fixed-point operand in Rx is equal to 1, otherwise cleared

Rn = LEFTZ Rx

Function

Extracts the number of leading 0s from the fixed-point operand in Rx. The extracted number is placed in the bit6 field in Rn.

Status Flags

SZ	Set if the MSB of Rx is 1, otherwise cleared
SV	Set if the result is 32, otherwise cleared
SS	Cleared

Rn = LEFTO Rx

Function

Extracts the number of leading 1s from the fixed-point operand in Rx. The extracted number is placed in the bit6 field in Rn.

Status Flags

SZ	Set if the MSB of Rx is 0, otherwise cleared
SV	Set if the result is 32, otherwise cleared
SS	Cleared

Rn = FPACK Fx

Function

Converts the IEEE 32-bit floating-point value in Fx to a 16-bit floating-point value stored in Rn. The short float data format has an 11-bit mantissa with a four-bit exponent plus sign bit. The 16-bit floating-point numbers reside in the lower 16 bits of the 32-bit floating-point field.

The result of the FPACK operation is:

$135 < \exp$[1]	Largest magnitude representation
$120 < \exp \leq 135$	Exponent is MSB of source exponent concatenated with the three LSBs of source exponent; the packed fraction is the rounded upper 11 bits of the source fraction
$109 < \exp \leq 120$	Exponent=0; packed fraction is the upper bits (source exponent – 110) of the source fraction prefixed by zeros and the “hidden” 1; the packed fraction is rounded
$\exp < 110$	Packed word is all zeros

1 exp = source exponent sign bit remains the same in all cases

The short float type supports gradual underflow. This method sacrifices precision for dynamic range. When packing a number which would have underflowed, the exponent is set to zero and the mantissa (including “hidden” 1) is right-shifted the appropriate amount. The packed result is a denormal which can be unpacked into a normal IEEE floating-point number.

Status Flags

SZ	Cleared
SV	Set if overflow occurs, cleared otherwise
SS	Cleared

Fn = FUNPACK Rx

Function

Converts the 16-bit floating-point value in Rx to an IEEE 32-bit floating-point value stored in Fx.

Result

$0 < \text{exp}^1 \leq 15$	Exponent is the three LSBs of the source exponent prefixed by the MSB of the source exponent and four copies of the complement of the MSB; the unpacked fraction is the source fraction with 12 zeros appended
exp = 0	Exponent is (120 – N) where N is the number of leading zeros in the source fraction; the unpacked fraction is the remainder of the source fraction with zeros appended to pad it and the "hidden" 1 stripped away

1 exp = source exponent sign bit remains the same in all cases

The short float type supports gradual underflow. This method sacrifices precision for dynamic range. When packing a number that would have underflowed, the exponent is set to 0 and the mantissa (including "hidden" 1) is right-shifted the appropriate amount. The packed result is a denormal, which can be unpacked into a normal IEEE floating-point number.

Status Flags

SZ	Cleared
SV	Cleared
SS	Cleared

Multifunction Computations

Multifunction computations are operations that occur simultaneously within the processor's computational unit. The syntax for these operations consists of combinations of instructions, delimited with commas and ended with a semicolon. The three types of multifunction computations appear below. Each type has a different format for the compute field.

- "Parallel Add and Subtract" on page 9-93
- "Parallel Multiplier and ALU" on page 9-95
- "Parallel Multiplier With Add and Subtract" on page 9-98

Operand Constraints

Each of the four input operands for multifunction computations are constrained to a different set of four register file locations, as shown in Figure 9-4. For example, the x-input to the ALU must be `R8`, `R9`, `R10`, or `R11`. In all other compute operations, the input operands can be any register file location.

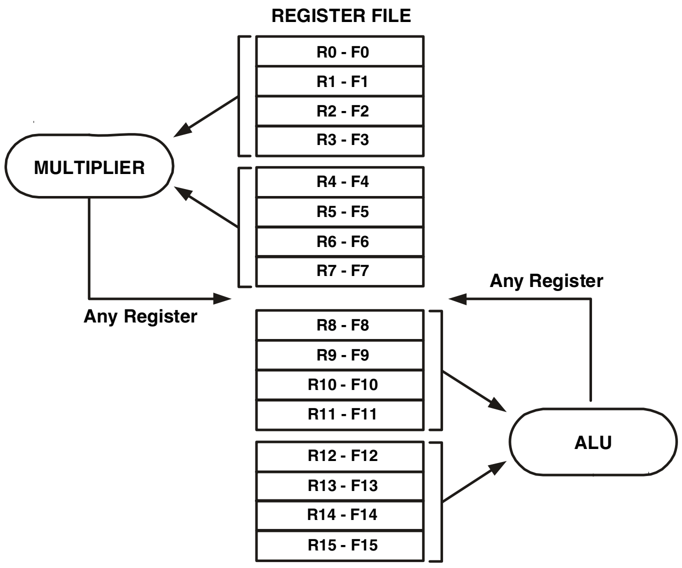


Figure 9-4. Permitted Input Registers for Multifunction Computations

Parallel Add and Subtract

Function (Fixed-Point)

Completes a dual add/subtract of the fixed-point fields in registers Rx and Ry. The sum is placed in the fixed-point field of register Ra and the difference in the fixed-point field of Rs. The floating-point extension fields of Ra and Rs are set to all 0s. In saturation mode (the ALU saturation mode bit in `MODE1` set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Function (Floating-Point)

Completes a dual add/subtract of the floating-point operands in registers Fx and Fy. The normalized results are placed in registers Fa and Fs: the sum in Fa and the difference in Fs. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in `MODE1`. Post-rounded overflow returns ±infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±zero. Denormal inputs are flushed to ±zero. A NAN input returns an all 1s result.

Syntax

Table 9-9 shows the fixed-point and floating-point syntax for multifunction add and subtract instructions.

Table 9-9. Multifunction, Parallel Add and Subtract

Syntax	Opcode (Bits 19–16)
Ra = Rx + Ry, Rs = Rx – Ry	0111
Fa = Fx + Fy, Fs = Fx – Fy	1111

Compute Field (Fixed-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	00		0111				Rs				Ra				Rx				Ry			

AZ	Set if an output is 0s, otherwise cleared
AU	Cleared
AN	Set if the most significant output bit is 1 for either of the outputs, otherwise cleared
AV	Set if the XOR of the carries of the two most significant adder stages of either of the outputs is 1, otherwise cleared
AC	Set if the carry from the most significant adder stage for either of the outputs is 1, otherwise cleared
AS	Cleared
AI	Cleared

Compute Field (Fixed-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	00		1111				Fs				Fa				Fx				Fy			

AZ	Set if either of the post-rounded results is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU	Set if either post-rounded result is a denormal, otherwise cleared
AN	Set if either of the floating-point results is negative, otherwise cleared
AV	Set if a post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC	Cleared
AS	Cleared
AI	Set if an input is a NAN or if both inputs are Infinities, otherwise cleared

Parallel Multiplier and ALU

Function

The parallel multiplier/ALU operation performs a multiply or multiply/accumulate and one of the following ALU operations: add, subtract, average, fixed-point to floating-point conversion or floating-point to fixed-point conversion, and/or floating-point abs, min, or max.

The multiplier and ALU operations are determined by `OPCODE`. The selections for the 6-bit `OPCODE` field are listed in Table 9-11. The multiplier x and y operands are received from data registers RXM (FXM) and RYM (FYM). The multiplier result operand is returned to data register RM (FM). The ALU x and y operands are received from data registers RXA (FXA) and RYA (FYA). The ALU result operand is returned to data register RA (FA).

The result operands can be returned to any registers within the register file. Each of the four input operands is restricted to a particular set of four data registers as shown in Table 9-10.

Table 9-10. Valid Data Registers for Input Operands

Input	Valid Sources
Multiplier X	R3-R0 (F3-F0)
Multiplier Y	R7-R4 (F7-F4)
ALU X	R11-R8 (F11-F8)
ALU Y	R15-R12 (F15-F12)

Syntax

Table 9-11 provides the syntax and opcode for each of the parallel multiplier and ALU instructions for both fixed-point and floating-point versions.

Table 9-11. Multifunction, Multiplier and ALU

Syntax	Opcode (Bits 22–16)
Rm = R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12	1000100
Rm = R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12	1000101
Rm = R3-0 * R7-4 (SSFR), Ra = (R11-8 + R15-12)/2	1000110
MRF = MRF + R3-0 * R7-4 (SSF), Ra = R11-8 + R15-12	1001000
MRF = MRF + R3-0 * R7-4 (SSF), Ra = R11-8 – R15-12	1001001
MRF = MRF + R3-0 * R7-4 (SSF), Ra = (R11-8 + R15-12)/2	1001010
Rm = MRF + R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12	1001100
Rm = MRF + R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12	1001101
Rm = MRF + R3-0 * R7-4 (SSFR), Ra =(R11-8 + R15-12)/2	1001110
MRF = MRF – R3-0 * R7-4 (SSF), Ra = R11-8 + R15-12	1010000
MRF = MRF – R3-0 * R7-4 (SSF), Ra = R11-8 – R15-12	1010001
MRF = MRF – R3-0 * R7-4 (SSF), Ra = (R11-8 + R15-12)/2	1010010
Rm = MRF – R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12	1010100
Rm = MRF – R3-0 * R7-4 (SSFR), Ra = R11-8 – R15-12	1010101
Rm = MRF – R3-0 * R7-4 (SSFR), Ra =(R11-8 + R15-12)/2	1010110
Fm = F3-0 * F7-4, Fa = F11-8 + F15-12	1011000
Fm = F3-0 * F7-4, Fa = F11-8 – F15-12	1011001
Fm = F3-0 * F7-4, Fa = FLOAT R11-8 by R15-12	1011010
Fm = F3-0 * F7-4, Fa = FIX F11-8 by R15-122	1011011
Fm = F3-0 * F7-4, Fa = ABS F11-8	1011101
Fm = F3-0 * F7-4, Fa = MAX (F11-8, F15-12)	1011110
Fm = F3-0 * F7-4, Fa = MIN (F11-8, F15-12)	1011111

Compute Field (Fixed-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Opcode (Table 9-11)						Rs				Ra				Rxm		Rym		Rxa		Rya	

Compute Field (Floating-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Opcode (Table 9-11)						Fs				Fa				Fxm		Fym		Fxa		Fya	

Parallel Multiplier With Add and Subtract

Function

The parallel multiplier and dual add/subtract operation performs a multiply or multiply/accumulate and computes the sum and the difference of the ALU inputs.

The multiplier x and y operands are received from data registers RXM (FXM) and RYM (FYM). The multiplier result operand is returned to data register RM (FM). The ALU x and y operands are received from data registers RXA (FXA) and RYA (FYA). The ALU result operands are returned to data register RA (FA) and RS (FS).

The result operands can be returned to any registers within the register file. Each of the four input operands is restricted to a different set of four data registers as shown in Table 9-12.

Table 9-12. Valid Sources of the Input Operands

Input	Valid Sources
Multiplier X	R3-R0 (F3-F0)
Multiplier Y	R7-R4 (f7-f4)
ALU X	R11-R8 (F11-F8)
ALU Y	R15-R12 (F15-F12)

Syntax

Table 9-13 provides the syntax and opcode for each of the parallel multiplier and add/subtract instructions for both fixed-point and floating-point versions.

Table 9-13. Multifunction, Multiplier and Dual Add and Subtract

Syntax	**Opcode (Bits 22–20)**
Rm=R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12, Rs=R11-8 – R15-12	110
Fm=F3-0 * F7-4, Fa=F11-8 + F15-12, Fs=F11-8 – F15-12	111

Compute Field (Fixed-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	10		Rs				Rm				Ra				RxmM		Rym		Rxa		Rya	

Compute Field (Floating-Point)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	11		Fs				Fm				Fa				Fxm		Fym		Fxa		Fya	

A INSTRUCTION SET QUICK REFERENCE

This instruction set summary provides a syntax summary for each instruction and includes a cross reference to each instruction's reference page.

Chapter Overview

The following summary topics appear in this chapter.

Compute and Move/Modify Summary

Compute and move/modify instructions are classed as Group I instructions. They provide math, conditional, and memory/register access services. The series of tables that follow summarize the Group I instructions. For a complete description of these instructions, see the noted pages.

"Type 1: Compute, Dreg«···»DM | Dreg«···»PM" on page 8-3

```
compute   | , DM(Ia, Mb) = dreg1   | | , PM(Ic, Md) = dreg2   | ;
          | , dreg1 = DM(Ia, Mb)   | | , dreg2 = PM(Ic, Md)   |
```

"Type 2: Compute" on page 8-6

```
IF  COND compute  ;
```

"Type 3: Compute, ureg«···»DM | PM, register modify" on page 8-8

```
IF COND compute   | , DM(Ia, Mb)   | =  ureg (LW);
                  | , PM(Ic, Md)   |

                  | , DM(Mb, Ia)   | =  ureg (LW);
                  | , PM(Md, Ic)   |

                    , ureg =       | DM(Ia, Mb) (LW);  |
                                   | PM(Ic, Md) (LW);  |

                    , ureg =       | DM(Mb, Ia) (LW);  |
                                   | PM(Md, Ic) (LW);  |
```

“Type 4: Compute, dreg«···»DM | PM, data modify” on page 8-13

```
IF COND compute        | , DM(Ia, <data6>)    |  = dreg ;
                       | , PM(Ic, <data6>)    |

                       | , DM(<data6>, Ia)    | = dreg ;
                       | , PM(<data6>, Ic)    |

                         , dreg  =            | DM(Ia, <data6>)  ;    |
                                              | PM(Ic, <data6>)  ;    |

                         , dreg  =            | DM(<data6>, Ia)  ;    |
                                              | PM(<data6>, Ic)  ;    |
```

“Type 5: Compute, ureg«··· »ureg | Xdreg<->Ydreg” on page 8-18

```
IF COND compute,       | ureg1 = ureg2        |  ;

                       | X dreg <-> Y dreg    |
```

“Type 6: Immediate Shift, dreg«···»DM | PM” on page 8-22

```
IF COND shiftimm       | , DM(Ia, Mb)         |  =  dreg  ;
                       | , PM(Ic, Md)         |

                         , dreg  =            | DM(Ia, Mb)  ;    |
                                              | PM(Ic, Md)  ;    |
```

“Type 7: Compute, modify” on page 8-27

```
IF COND   compute        , MODIFY             | (Ia, Mb)  ;    |
                                              | (Ic, Md)  ;    |
```

Program Flow Control Summary

Program flow control instructions are classed as Group II instructions and These instructions control program execution flow. The series of tables that follow summarize the Group II instructions. For a complete description of these instructions, see the noted pages.

“Type 8: Direct Jump | Call” on page 8-31

IF COND JUMP	<addr24>	(DB)	;
	(PC, <reladdr24>)	(LA)	
		(CI)	
		(DB, LA)	
		(DB, CI)	

IF COND CALL	<addr24>	(DB) ;
	(PC, <reladdr24>)	

“Type 9: Indirect Jump | Call, Compute” on page 8-35

IF COND JUMP	(Md, Ic)	(DB)	, compute	;
	(PC, <reladdr6>)	(LA)	, ELSE compute	
		(CI)		
		(DB, LA)		
		(DB, CI)		

IF COND CALL	(Md, Ic)	(DB)	, compute	;
	(PC, <reladdr6>)		, ELSE compute	

“Type 10: Indirect Jump | Compute, dreg«···»DM” on page 8-42

IF COND Jump	(Md, Ic)	, Else	compute, DM(Ia, Mb) = dreg ;
	(PC, <reladdr6>)		compute, dreg = DM(Ia, Mb) ;

“Type 11: Return From Subroutine | Interrupt, Compute” on page 8-48

IF COND RTS	(DB)	, compute	;
	(LR)	, ELSE compute	
	(DB, LR)		

IF COND RTI	(DB)	, compute	;
		, ELSE compute	

“Type 12: Do Until Counter Expired” on page 8-53

LCNTR =	<data16>	, DO	<addr24>	UNTIL LCE;
	ureg		(PC, <reladdr24>)	

“Type 13: Do Until” on page 8-55

DO	<addr24>	UNTIL termination ;
	(PC, <reladdr24>)	

Immediate Move Summary

Immediate move instructions are classed as Group III instructions. They provide memory/register access services. The series of tables that follow summarize the Group III instructions. For a complete description of these instructions, see the noted pages.

"Type 14: Ureg«···»DM | PM (direct addressing)" on page 8-59

```
| DM(<addr32>)          | = ureg (LW);
| PM(<addr32>)          |

ureg =                  | DM(<addr32>) (LW);    |
                        | PM(<addr32>) (LW);    |
```

"Type 15: Ureg«···»DM | PM (indirect addressing)" on page 8-62

```
| DM(<data32>, Ia)      | = ureg                  (LW);
| PM(<data32>, Ic)      |

ureg =                  | DM(<data32>, Ia)      | (LW);
                        | PM(<data32>, Ic)      |
```

"Type 16: Immediate data···»DM | PM" on page 8-66

```
| DM(Ia, Mb)            | = <data32> ;
| PM(Ic, Md)            |
```

"Type 17: Immediate data···»Ureg" on page 8-69

```
ureg = <data32> ;
```

Miscellaneous Operations Summary

Miscellaneous instructions are classed as Group IV instructions. They provide system register, bit manipulation, and low power services. The series of tables that follow summarize the Group IV instructions. For a complete description of these instructions, see the noted pages.

The Type 23: IDLE16 and the Type 24: creg<<--->>ureg instructions are not supported on the ADSP-2136x processors.

"Type 18: System Register Bit Manipulation" on page 8-72

BIT	SET	sreg <data32> ;
	CLR	
	TGL	
	TST	
	XOR	

"Type 19: I Register Modify | Bit-Reverse" on page 8-75

MODIFY	(Ia, <data32>)	;
	(Ic, <data32>)	

BITREV	(Ia, <data32>)	;
	(Ic, <data32>)	

"Type 20: Push, Pop Stacks, Flush Cache" on page 8-78

PUSH	LOOP ,	PUSH	STS ,	PUSH	PCSTK , FLUSH CACHE ;
POP		POP		POP	

"Type 21: Nop" on page 8-80

NOP ;

"Type 22: Idle" on page 8-81

IDLE ;

"Type 25: Cjump/Rframe" on page 8-82

CJUMP	function	(DB) ;
	(PC, <reladdr24>)	

RFRAME ;

Register Types Summary

Table A-1 and Table A-2 list ADSP-2136x processor registers. The registers in Table A-1 are in the core processor and the registers in Table A-2 are in the integrated I/O processor sections of the processor.

Table A-1. Universal Registers (Ureg)

Register Type	Register(s)	Function
Register File (ureg & dreg)	R0 – R15	Processing element X register file locations, fixed-point
	F0 – F15	Processing element X register file locations, floating-point
	S0 – S15	Processing element Y register file locations, fixed-point
	SF0 – SF15	Processing element Y register file locations, floating-point
Program Sequencer	PC	Program counter (read-only)
	PCSTK	Top of PC stack
	PCSTKP	PC stack pointer
	FADDR	Fetch address (read-only)
	DADDR	Decode address (read-only)
	LADDR	Loop termination address, code; top of loop address stack
	CURLCNTR	Current loop counter; top of loop count stack
	LCNTR	Loop count for next nested counter-controlled loop

Table A-1. Universal Registers (Ureg) (Cont'd)

Register Type	Register(s)	Function
Data Address Generators	I0 – I7	DAG1 index registers
	M0 – M7	DAG1 modify registers
	L0 – L7	DAG1 length registers
	B0 – B7	DAG1 base registers
	I8 – I15	DAG2 index registers
	M8 – M15	DAG2 modify registers
	L8 – L15	DAG2 length registers
	B8 – B15	DAG2 base registers
Bus Exchange	PX1	PMD-DMD bus exchange 1 (32 bits)
	PX2	PMD-DMD bus exchange 2 (32 bits)
	PX	64-bit combination of PX1 and PX2
Timer	TPERIOD	Timer period
	TCOUNT	Timer counter

Table A-1. Universal Registers (Ureg) (Cont'd)

Register Type	Register(s)	Function
System Registers (sreg & ureg)	MODE1	Mode control & status
	MODE2	Mode control & status
	IRPTL	Interrupt latch
	IMASK	Interrupt mask
	IMASKP	Interrupt mask pointer (for nesting)
	MMASK	Mode mask
	FLAGS	Flag pins input/output state
	LIRPTL	Link Port interrupt latch, mask, and pointer
	ASTATx	Element x arithmetic status flags, bit test flag, etc.
	ASTATy	Element y arithmetic status flags, bit test flag, etc.
	STKYx	Element x sticky arithmetic status flags, stack status flags, and so on.
	STKYy	Element y sticky arithmetic status flags, stack status flags, and so on.
	USTAT1	User status register 1
	USTAT2	User status register 2
	USTAT3	User status register 3
	USTAT4	User status register 4

Table A-2. I/O and Multiplier Registers

Register Type	Register(s)	Function
IOP Registers (system control)	SYSCON	System control
	SYSTAT	System status
	WAIT	Memory wait states
	VIRPT	Multiprocessor IRQ
IOP Registers (system control)	For a complete list of IOP registers, see Appendix A, “Input/Output Registers” in the processor specific *ADSP-2136x SHARC Processor Hardware Reference*. Specifically, Table A-1, I/O Processor Register Groups, provides a comprehensive listing of all IOP registers.	
Multiplier Registers	MR, MR0, MR1, MR2,	Multiplier results
	MRF, MR0F, MR1F, MR2F	Multiplier results, foreground
	MRB, MR0B, MR1B, MR2B	Multiplier results, background

Memory Addressing Summary

ADSP-2136x processors support the following types of addressing.

Direct Addressing

Absolute address (Instruction Types 8, 12, 13, 14)

```
dm(0x000015F0) = astat;

if ne jump label2;         {'label2' is an address label}
```

PC-relative address (Instruction Types 8, 9, 10, 12, 13)

```
call(pc,10), r0=r6+r3;

do(pc,length) until sz;    {'length' is a variable}
```

Indirect Addressing (using DAG registers):

Post-modify with M register, update I register (Instruction Types 1, 3, 6, 16)

```
f5=pm(i9,m12);

dm(i0,m3)=r3, r1=pm(i15,m10);
```

Pre-modify with M register, no update (Instruction Types 3, 9, 10)

```
r1=pm(m10,i15);

jump(m13,i11);
```

Post-modify with immediate value, update I register (Instruction Type 4)

```
f15=dm(i0,6);
```

```
if av r1=pm(i15,0x11);
```

Pre-modify with immediate value, no update (Instruction Types 4, 15)

```
if av r1=pm(0x11,i15);
```

```
dm(127,i5)=laddr;
```

Instruction Set Notation Summary

The conventions for ADSP-2136x instruction syntax descriptions appear in Table A-3. Other parts of the instruction syntax and opcode information also appear in this section.

Table A-3. Instruction Set Notation

Notation	Meaning
UPPERCASE	Explicit syntax—assembler keyword (notation only; assembler is case-insensitive and lowercase is the preferred programming convention)
;	Semicolon (instruction terminator)
,	Comma (separates parallel operations in an instruction)
italics	Optional part of instruction
| option1 | | option2 |	List of options between vertical bars (choose one)
compute	ALU, multiplier, shifter or multifunction operation (see “Computations Reference” on page 9-1)

Table A-3. Instruction Set Notation (Cont'd)

Notation	Meaning
shiftimm	Shifter immediate operation (see "Computations Reference" on page 9-1)
cond	Status condition (see condition codes in Table A-4 on page A-16)
termination	Loop termination condition (see condition codes in Table A-4 on page A-16)
ureg	Universal register
cureg	Complementary universal register (see Table A-10 on page A-25)
sreg	System register
csreg	Complementary system register (see Table A-10 on page A-25)
dreg	Data register (register file): R15-R0 or F15-F0
cdreg	Complementary data register (register file): R15-R0 or F15-F0 (see Table A-10 on page A-25)
creg	One of 32 cache entries, an entry consisting of a CH, CL, & CA
Ia	I7-I0 (DAG1 index register)
Mb	M7-M0 (DAG1 modify register)
Ic	I15-I8 (DAG2 index register)
Md	M15-M8 (DAG2 modify register)
<datan>	n-bit immediate data value
<addrn>	n-bit immediate address value
<reladdrn>	n-bit immediate PC-relative address value
+1	the incremented data, address or register value
(DB)	Delayed branch
(LA)	Loop abort (pop loop and PC stacks on branch)
(CI)	Clear interrupt
(LR)	Loop reentry
(LW)	Long Word (forces long word access in normal word range)

Conditional Execution Codes Summary

In a conditional instruction, execution of the entire instruction depends on the specified `condition` (`cond` or `terminate`). Table A-4 lists the codes for conditionals use (IF and DO UNTIL).

Table A-4. IF Condition and Do/Until Termination Mnemonics

Condition From	Description	True If…	Mnemonic
ALU	ALU = 0	AZ = 1	EQ
	ALU ≠ 0	AZ = 0	NE
	ALU > 0	footnote[1]	GT
	ALU < zero	footnote[2]	LT
	ALU ≥ 0	footnote[3]	GE
	ALU ≤ 0	footnote[4]	LE
	ALU carry	AC = 1	AC
	ALU not carry	AC = 0	NOT AC
	ALU overflow	AV = 1	AV
	ALU not overflow	AV = 0	NOT AV
Multiplier	Multiplier overflow	MV = 1	MV
	Multiplier not overflow	MV= 0	NOT MV
	Multiplier sign	MN = 1	MS
	Multiplier not sign	MN = 0	NOT MS
Shifter	Shifter overflow	SV = 1	SV
	Shifter not overflow	SV = 0	NOT SV
	Shifter zero	SZ = 1	SZ
	Shifter not zero	SZ = 0	NOT SZ
Bit Test	Bit test flag true	BTF = 1	TF
	Bit test flag false	BTF = 0	NOT TF

Table A-4. IF Condition and Do/Until Termination Mnemonics (Cont'd)

Condition From	Description	True If…	Mnemonic
Flag Input	Flag0 asserted	FI0 = 1	FLAG0_IN
	Flag0 not asserted	FI0 = 0	NOT FLAG0_IN
	Flag1 asserted	FI1 = 1	FLAG1_IN
	Flag1 not asserted	FI1 = 0	NOT FLAG1_IN
	Flag2 asserted	FI2 = 1	FLAG2_IN
	Flag2 not asserted	FI2 = 0	NOT FLAG2_IN
	Flag3 asserted	FI3 = 1	FLAG3_IN
	Flag3 not asserted	FI3 = 0	NOT FLAG3_IN
Mode	Bus master true		BM
	Bus master false		NOT BM
Sequencer	Loop counter expired (Do)	CURLCNTR = 1	LCE
	Loop counter not expired (If)	CURLCNTR ≠ 1	NOT ICE
	Always false (Do)	Always	FOREVER
	Always true (If)	Always	TRUE

1 ALU greater than (GT) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN)] or $\overline{\text{AZ}}$ = 0

2 ALU less than (LT) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN and $\overline{\text{AZ}}$)] = 1

3 ALU greater equal (GE) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN and $\overline{\text{AZ}}$)] = 0

4 ALU lesser or equal (LT) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN)] or $\overline{\text{AZ}}$ = 1

SISD/SIMD Conditional Testing Summary

The processor handles conditional execution differently in SISD versus SIMD mode. There are three ways that conditionals differ in SIMD mode:

- In conditional computation (if ... compute) instructions, each processing element executes the computation based on evaluating the condition in that processing element.
- In conditional program control (if ... jump/call) instructions, the program sequencer executes the jump/call based on a logical AND of the conditions in both processing elements.
- In conditional computation instructions with an ELSE clause, each processing element executes the ELSE computation based on evaluating the inverse of the condition (not cond) in that processing element.

Table A-5 and Table A-6 compare SISD and SIMD if-ELSE conditional execution, which are available in the Type 9, 10, and 11 instructions.

Table A-5. SISD Mode Conditional Execution

Conditional Test	ELSE Modifier	Results for Type 11 (RTS)
0 (false)	0 (without else)	rts nops, compute nops
0 (false)	1 (else)	rts nops, compute executes
1 (true)	0 (without else)	rts executes, compute executes
1 (true)	1 (else)	rts executes, compute nops

Table A-6. SIMD Mode Conditional Execution

Conditional Test		Else Modifier	Results for Type 11 (RTS)
PEx	PEy		
0	0	0	rts nops, pex compute nops, pey compute nops
0	1	0	rts nops, pex compute nops, pey compute executes
1	0	0	rts nops, pex compute exe, pey compute nops
1	1	0	rts exe, pex compute exe, pey compute exe
0	0	1	rts nops, pex compute exe, pey compute exe
0	1	1	rts nops, pex compute exe, pey compute nops
1	0	1	rts nops, pex compute nops, pey compute exe
1	1	1	rts exe, pex compute nops, pey compute nops

For more information and examples, see the following instruction reference pages.

- "Type 9: Indirect Jump | Call, Compute" on page 8-35
- "Type 10: Indirect Jump | Compute, dreg«···»DM" on page 8-42
- "Type 11: Return From Subroutine | Interrupt, Compute" on page 8-48

Instruction Opcode Acronym Summary

In ADSP-2136x processor opcodes, some bits are explicitly defined to be zeros or ones. The values of other bits or fields set various parameters for the instruction. The terms in Table A-7 define these opcode bits and fields. Unspecified bits are ignored when the processor decodes the instruction, but are reserved for future use.

Table A-7. Opcode Acronyms

Bit/Field	Description	States
A	Loop abort code	0 Do not pop loop, PC stacks on branch 1 Pop loop, PC stacks on branch
ADDR	Immediate address field	
AI	Computation unit register	0000 MR0F 0001 MR1F 0010 MR2F 0100 MR0B 0101 MR1B 0110 MR2B
B	Branch type	0 Jump 1 Call
BOP	Bit operation select codes	000 Set 001 Clear 010 Toggle 100 Test 101 XOR
COMPUTE	Compute operation field (see "Computations Reference" in Chapter 9, Computations Reference)	
COND	Status condition codes	0–31
CI	Clear interrupt code	0 Do not clear current interrupt 1 Clear current interrupt
CREG	Instruction cache entry	0–31
CS	Instruction cache register select code	00 Lower half of instruction RAM entry 01 Upper half of instruction RAM entry 11 Address CAM entry
CU	Computation unit select codes	00 ALU 01 Multiplier 10 Shifter
DATA	Immediate data field	
DEC	Counter decrement code	0 No counter decrement 1 Counter decrement

Table A-7. Opcode Acronyms (Cont'd)

Bit/Field	Description	States
DMD	Memory access direction	0 Read 1 Write
DMI	Index (I) register numbers, DAG1	0–7
DMM	Modify (M) register numbers, DAG1	0–7
DREG	Register file locations	0–15
E	ELSE clause code	0 No ELSE clause 1 ELSE clause
FC	Flush cache code	0 No cache flush 1 Cache flush
G	DAG/Memory select	0 DAG1 or data memory 1 DAG2 or program memory
INC	Counter increment code	0 No counter increment 1 Counter increment
J	Jump type	0 Non-delayed 1 Delayed
L	Long word memory address	0 Access size based on memory map 1 Long word (64-bit) access size
LPO	Loop stack pop code	0 No stack pop 1 Stack pop
LPU	Loop stack push code	0 No stack push 1 Stack push
LR	Loop reentry code	0 No loop reentry 1 Loop reentry
NUM	Interrupt vector	0–7
PMD	Memory access direction	0 Read 1 Write
PMI	Index (I) register numbers, DAG2	8–15

Table A-7. Opcode Acronyms (Cont'd)

Bit/Field	Description	States
PMM	Modify (M) register numbers, DAG2	8–15
PPO	PC stack pop code	0 No stack pop 1 Stack pop
PPU	PC stack push code	0 No stack push 1 Stack push
RELADDR	PC-relative address field	
S	UREG transfer/instruction cache read-load select	0 Instruction cache read-load 1 Ureg transfer
SPO	Status stack pop code	0 No stack pop 1 Stack pop
SPU	Status stack push code	0 No stack push 1 Stack push
SREG	System register code	0–15 (see "Universal Register Codes" on page A-23)
TERM	Termination condition codes	0–31
U	Update, index (I) register	0 Pre-modify, no update 1 Post-modify with update
UREG	Universal register code	0–256 (see "Universal Register Codes" on page A-23)
RA, RM, RN, RS, RX, RY	Register file locations for compute operands and results	0–15
RXA	ALU x-operand register file location for multifunction operations	8–11
RXM	Multiplier x-operand register file location for multifunction operations	0–3

Table A-7. Opcode Acronyms (Cont'd)

Bit/Field	Description	States
RYA	ALU y-operand register file location for multifunction operations	12–15
RYM	Multiplier y-operand register file location for multifunction operations	4–7

Universal Register Codes

Table A-8, Table A-9, Table A-10, and Table A-11 in this section list the bit codes for registers that appear within opcode fields.

Table A-8. Universal Registers

Register	Description
PC	program counter
PCSTK	top of PC stack
PCSTKP	PC stack pointer
FADDR	fetch address
DADDR	decode address
LADDR	loop termination address
CURLCNTR	current loop counter
LCNTR	loop counter
R15–R0	X element register file locations
S15–S0	Y element register file locations
I15–I0	DAG1 and DAG2 index registers
M15–M0	DAG1 and DAG2 modify registers
L15–L0	DAG1 and DAG2 length registers

Table A-8. Universal Registers (Cont'd)

Register	Description
B15–B0	DAG1 and DAG2 base registers
PX	48-bit PX1 and PX2 combination
PX1	bus exchange 1 (16 bits)
PX2	bus exchange 2 (32 bits)
TPERIOD	timer period
TCOUNT	timer counter

Table A-9. Universal and System Registers

Register	Description
MODE1	mode control 1
MODE2	mode control 2
IRPTL	interrupt latch
IMASK	interrupt mask
IMASKP	interrupt mask pointer
MMASK	mode mask
FLAGS	flag pins input/output state
ASTATx	X element arithmetic status
STKYx	X element sticky status
ASTATy	Y element arithmetic status
STKYy	Y element sticky status
USTAT1	user status reg 1
USTAT2	user status reg 2
USTAT3	user status reg 3
USTAT4	user status reg 4

Table A-10. Complementary Registers (Ureg–Cureg)

Register Type	SIMD Mode Complementary Registers
Data Register (Dreg and Ureg)	R0–S0 R1–S1 R2–S2 R3–S3 R4–S4 R5–S5 R6–S6 R7–S7 R8–S8 R9–S9 R10–S10 R11–S11 R12–S12 R13–S13 R14–S14 R15–S15
System Register (Sreg and Ureg)	USTAT1–USTAT2 USTAT3–USTAT4 ASTATx–ASTATy STKYx–STKYy
Bus Exchange Register (Ureg)	PX1–PX2

Table A-11 shows how the `Ureg` register codes appear to PEx.

Table A-11. Processing Element X Universal Register Codes (SISD/SIMD)

Bits: *3210* ⇩	**Bits:** *7654* **0000**	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**
0000	R0	I0	M0	L0	B0	S0	FADDR	USTAT1
0001	R1	I1	M1	L1	B1	S1	DADDR	USTAT2
0010	R2	I2	M2	L2	B2	S2		MODE1

Table A-11. Processing Element X Universal Register Codes (SISD/SIMD) (Cont'd)

Bits: *3210* ⇩	**Bits:** *7654* **0000**	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**
0011	R3	I3	M3	L3	B3	S3	PC	MMASK
0100	R4	I4	M4	L4	B4	S4	PCSTK	MODE2
0101	R5	I5	M5	L5	B5	S5	PCSTKP	FLAGS
0110	R6	I6	M6	L6	B6	S6	LADDR	ASTATx
0111	R7	I7	M7	L7	B7	S7	CURL-CNTR	ASTATy
1000	R8	I8	M8	L8	B8	S8	LCNTR	STKYx
1001	R9	I9	M9	L9	B9	S9	EMUCLK	STKYy
1010	R10	I10	M10	L10	B10	S10	EMUCLK2	IRPTL
1011	R11	I11	M11	L11	B11	S11	PX	IMASK
1100	R12	I12	M12	L12	B12	S12	PX1	IMASKP
1101	R13	I13	M13	L13	B13	S13	PX2	LRPTL
1110	R14	I14	M14	L14	B14	S14	TPERIOD	USTAT3
1111	R15	I15	M15	L15	B15	S15	TCOUNT	USTAT4

Table A-12 shows how the *Ureg* register codes appear to PEy.

Table A-12. Processing Element Y Universal Register Codes (SIMD)

Bits: *3210* ⇩	**Bits:** *7654* **0000**	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**
0000	S0	I0	M0	L0	B0	R0	FADDR	USTAT2
0001	S1	I1	M1	L1	B1	R1	DADDR	USTAT1
0010	S2	I2	M2	L2	B2	R2		MODE1
0011	S3	I3	M3	L3	B3	R3	PC	MMASK
0100	S4	I4	M4	L4	B4	R4	PCSTK	MODE2
0101	S5	I5	M5	L5	B5	R5	PCSTKP	FLAGS
0110	S6	I6	M6	L6	B6	R6	LADDR	ASTATy
0111	S7	I7	M7	L7	B7	R7	CURL-CNTR	ASTATx
1000	S8	I8	M8	L8	B8	R8	LCNTR	STKYy
1001	S9	I9	M9	L9	B9	R9	EMUCLK	STKYx
1010	S10	I10	M10	L10	B10	R10	EMUCLK2	IRPTL
1011	S11	I11	M11	L11	B11	R11	PX	IMASK
1100	S12	I12	M12	L12	B12	R12	PX2	IMASKP
1101	S13	I13	M13	L13	B13	R13	PX1	LRPTL
1110	S14	I14	M14	L14	B14	R14	TPERIOD	USTAT4
1111	S15	I15	M15	L15	B15	R15	TCOUNT	USTAT3

ADSP-2136x Instruction Opcode Map

Table A-13. ADSP-2136x Processor Opcodes (Bits 47–27)

Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
"Type 1: Compute, Dreg«···»DM | Dreg«···»PM"	001			DMD	DMI			DMM			PMD	DM DREG				PMI			PMM		
"Type 2: Compute"	000			00001							COND										
"Type 3: Compute, ureg«···»DM | PM, register modify"	010			U	I			M			COND					G	D	L	UREG>		
"Type 4: Compute, dreg«···»DM | PM, data modify"	011			0	I			G	D	U	COND					DATA					
(a) "Type 5: Compute, ureg«··· »ureg | Xdreg<->Ydreg"	011			1	0	SRC UREG					COND					SU			DEST UREG>		
(b) "Type 5: Compute, ureg«··· »ureg | Xdreg<->Ydreg"	011			1	1		Y DREG				COND										
(a) "Type 6: Immediate Shift, dreg«···»DM | PM"	100			0	I			M			COND					G	D	DATAEX			
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Table A-14. ADSP-2136x Processor Opcodes (Bits 26–0)

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PM DREG				COMPUTE																						
				COMPUTE																						
<UREG				COMPUTE																						
DREG				COMPUTE																						
<DEST UREG				COMPUTE																						
X DREG				COMPUTE																						
DREG				0	SHIFTOP						DATA								RN				RX			
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table A-15. ADSP-2136x Processor Opcodes (Bits 47–27) (Cont'd)

Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
(b) "Type 6: Immediate Shift, dreg«···»DM | PM"	000			00010							COND							DATAEX			
"Type 7: Compute, modify"	000			00100						G	COND					I			M		
(a) "Type 8: Direct Jump | Call"	000			00110					B	A	COND										
(b) "Type 8: Direct Jump | Call"	000			00111					B	A	COND										
(a) "Type 9: Indirect Jump | Call, Compute"	000			01000					B	A	COND					PMI			PMM		
(b) "Type 9: Indirect Jump | Call, Compute"	000			01001					B	A	COND					RELADDR					
(a) "Type 10: Indirect Jump | Compute, dreg«···»DM"	110			D	DMI			DMM			COND					PMI			PMM		
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Table A-16. ADSP-2136x Processor Opcodes (Bits 26–0) (Cont'd)

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				0	SHIFTOP						DATA								RN				RX			
				COMPUTE																						
J		CI	ADDR																							
J		CI	RELADDR																							
J	E	CI		COMPUTE																						
J	E	CI		COMPUTE																						
DREG				COMPUTE																						
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table A-17. ADSP-2136x Processor Opcodes (Bits 47–27) (Cont’d)

Instruction Type	**47**	**46**	**45**	**44**	**43**	**42**	**41**	**40**	**39**	**38**	**37**	**36**	**35**	**34**	**33**	**32**	**31**	**30**	**29**	**28**	**27**
(b) “Type 10: Indirect Jump | Compute, dreg«···»DM”	111			D	DMI			DMM			COND					RELADDR					
(a) “Type 11: Return From Subroutine | Interrupt, Compute”	000			01010							COND										
(b) “Type 11: Return From Subroutine | Interrupt, Compute”	000			01011							COND										
(a) “Type 12: Do Until Counter Expired”	000			01100					DATA>												
(b) “Type 12: Do Until Counter Expired”	000			01101					0	UREG											
“Type 13: Do Until”	000			01110							TERM										
“Type 14: Ureg«···»DM | PM (direct addressing)”	000			100			G	D	L	UREG							ADDR (upper 5 bits)				
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Table A-18. ADSP-2136x Processor Opcodes (Bits 26–0) (Cont'd)

26	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
DREG				COMPUTE																						
J	E	L R		COMPUTE																						
J	E			COMPUTE																						
<DATA			RELADDR																							
			RELADDR																							
			RELADDR																							
ADDR (lower 27 bits)																										
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table A-19. ADSP-2136x Processor Opcodes (Bits 47–27) (Cont'd)

Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27
"Type 15: Ureg«···»DM | PM (indirect addressing)"	101			G	I			D	L	UREG							DATA (upper 5 bits)				
"Type 16: Immediate data···»DM | PM"	100			1	I			M			G						DATA (upper 5 bits)				
"Type 17: Immediate data···»Ureg"	000			01111					0	UREG							DATA (upper 5 bits)				
"Type 18: System Register Bit Manipulation"	000			10100					BOP				SREG				DATA (upper 5 bits)				
(a) "Type 19: I Register Modify | Bit-Reverse"	000			10110					0	G				I			DATA (upper 5 bits)				
(b)"Type 19: I Register Modify | Bit-Reverse"	000			10110					1	G				I			DATA (upper 5 bits)				
"Type 20: Push, Pop Stacks, Flush Cache"	000			10111					LPU	LPO	SPU	SPO	PPU	PPO	FC						
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Table A-20. ADSP-2136x Processor Opcodes (Bits 26–0) (Cont'd)

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
DATA (lower 27 bits)																										
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table A-21. ADSP-2136x Processor Opcodes (Bits 47–27) (Cont'd)

Instruction Type	**47**	**46**	**45**	**44**	**43**	**42**	**41**	**40**	**39**	**38**	**37**	**36**	**35**	**34**	**33**	**32**	**31**	**30**	**29**	**28**	**27**
"Type 21: Nop"	000			00000					0												
"Type 22: Idle"	000			00000					1												
(a) "Type 25: Cjump/Rframe"	0001				1000				0000				0100				0000				0
(b) "Type 25: Cjump/Rframe"	0001				1000				0100				0100				0000				0
(c) "Type 25: Cjump/Rframe"	0001				1001				0000				0000				0000				0
Instruction Type	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27

Table A-22. ADSP-2136x Processor Opcodes (Bits 26–0) (Cont'd)

26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
000			ADDR																							
000			RELADDR																							
000			0000				0000				0000				0000				0000				0000			
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

B REGISTERS

The ADSP-2136x processor has general-purpose and dedicated registers in each of its functional blocks. The register reference information for each functional block includes bit definitions, initialization values, and memory-mapped addresses (for I/O processor registers). Information on each type of register is available at the following locations:

- "Control and Status System Registers" on page B-2
- "Processing Element Registers" on page B-22
- "Program Sequencer Registers" on page B-24
- "Data Address Generator Registers" on page B-34
- "Timer Registers" on page B-35
- "Power Management Registers" on page B-38
- "I/O Processor Registers" on page B-43

When writing processor programs, it is often necessary to set, clear, or test bits in the processor's registers. While these bit operations can all be done by referring to the bit's location within a register or (for some operations) the register's address with a hexadecimal number, it is much easier to use symbols that correspond to the bit's or register's name. For convenience and consistency, Analog Devices provides a header file that contains these bit and registers definitions. An #include file is provided with the VisualDSP tools and can be found in the `VisualDSP/2136x/`include directory.

Many registers have reserved bits. When writing to a register, programs may only clear (write zero to) the register's reserved bits.

For more information, see "Interrupt Registers" in the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* and the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

Control and Status System Registers

The processor's control and status system registers determine how the processor core operates and indicate the status of many processor core operations. In "Instruction Set" in Chapter 8, Instruction Set, these registers are referred to as system registers (`Sreg`), which are a subset of the processor's universal registers (`Ureg`). Not all registers are valid in all assembly language instructions. In the assembly syntax descriptions, the register group name (`Ureg`, `Sreg`, and others) indicates which type of register is valid within the instruction's context. Table B-1 lists the processor core's control and status registers with their initialization values. Descriptions of each register follow. Other system registers (`Sreg`) are in the I/O processor. These registers are described in detail in the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* and the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

Table B-1. Control and Status Registers for the Processor Core

Register Name and Page Reference	Initialization After Reset
"Mode Control 1 Register (MODE1)" on page B-3	0x0000 0000
"Mode Mask Register (MMASK)" on page B-7	0x0020 0000
"Mode Control 2 Register (MODE2)" on page B-11	0x4200 0000
"Arithmetic Status Registers (ASTATx and ASTATy)" on page B-12	0x0000 0000
"Sticky Status Registers (STKYx and STKYy)" on page B-17	0x0540 0000
"User-Defined Status Registers (USTATx)" on page B-21	0x0000 0000

Mode Control 1 Register (MODE1)

The mode control 1 register is a non memory-mapped, universal, system register (`Ureg` and `Sreg`). The reset value for this register is 0x0000 0000. Figure B-1, Figure B-2, and Table B-3 provide bit information for the `MODE1` register.

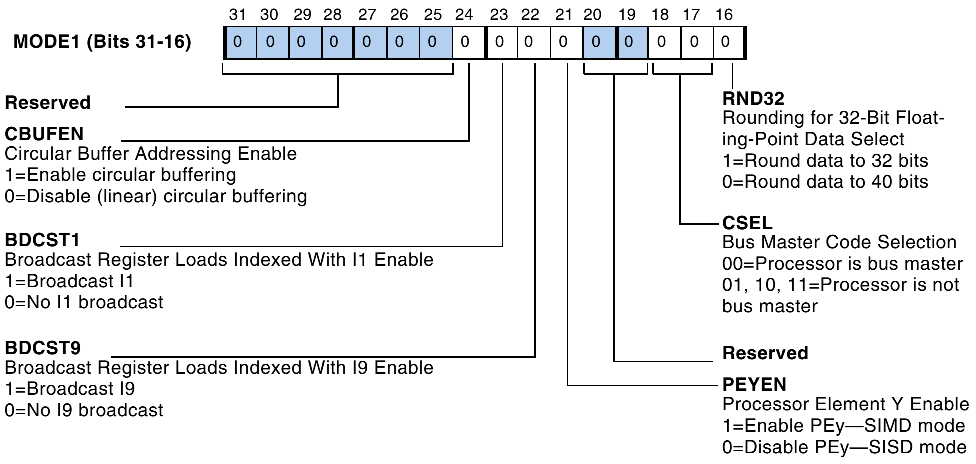

MODE1 (Bits 31-16)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reserved
CBUFEN
Circular Buffer Addressing Enable
1=Enable circular buffering
0=Disable (linear) circular buffering
BDCST1
Broadcast Register Loads Indexed With I1 Enable
1=Broadcast I1
0=No I1 broadcast
BDCST9
Broadcast Register Loads Indexed With I9 Enable
1=Broadcast I9
0=No I9 broadcast
RND32
Rounding for 32-Bit Floating-Point Data Select
1=Round data to 32 bits
0=Round data to 40 bits
CSEL
Bus Master Code Selection
00=Processor is bus master
01, 10, 11=Processor is not bus master
Reserved
PEYEN
Processor Element Y Enable
1=Enable PEy—SIMD mode
0=Disable PEy—SISD mode

Figure B-1. Mode Control 1 Register Bits 31–16

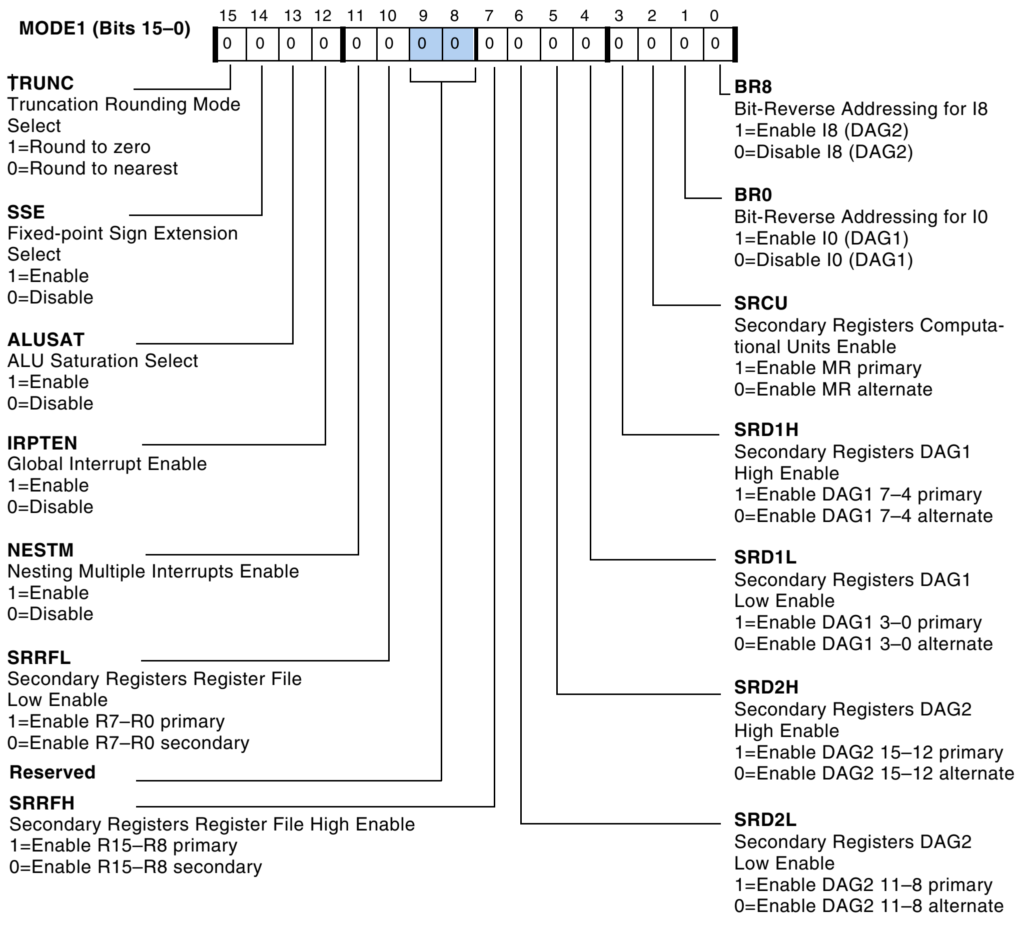

MODE1 (Bits 15–0)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
†TRUNC
Truncation Rounding Mode Select
1=Round to zero
0=Round to nearest
SSE
Fixed-point Sign Extension Select
1=Enable
0=Disable
ALUSAT
ALU Saturation Select
1=Enable
0=Disable
IRPTEN
Global Interrupt Enable
1=Enable
0=Disable
NESTM
Nesting Multiple Interrupts Enable
1=Enable
0=Disable
SRRFL
Secondary Registers Register File Low Enable
1=Enable R7–R0 primary
0=Enable R7–R0 secondary
Reserved
SRRFH
Secondary Registers Register File High Enable
1=Enable R15–R8 primary
0=Enable R15–R8 secondary
BR8
Bit-Reverse Addressing for I8
1=Enable I8 (DAG2)
0=Disable I8 (DAG2)
BR0
Bit-Reverse Addressing for I0
1=Enable I0 (DAG1)
0=Disable I0 (DAG1)
SRCU
Secondary Registers Computational Units Enable
1=Enable MR primary
0=Enable MR alternate
SRD1H
Secondary Registers DAG1 High Enable
1=Enable DAG1 7–4 primary
0=Enable DAG1 7–4 alternate
SRD1L
Secondary Registers DAG1 Low Enable
1=Enable DAG1 3–0 primary
0=Enable DAG1 3–0 alternate
SRD2H
Secondary Registers DAG2 High Enable
1=Enable DAG2 15–12 primary
0=Enable DAG2 15–12 alternate
SRD2L
Secondary Registers DAG2 Low Enable
1=Enable DAG2 11–8 primary
0=Enable DAG2 11–8 alternate

Figure B-2. Mode Control 1 Register Bits 15–0

Table B-2. Mode Control 1 Register (MODE1) Bit Descriptions

Bit	Name	Description
0	BR8	**Bit-Reverse Addressing For Index I8 Enable.** Enables (bit reversed if set, = 1) or disables (normal if cleared, = 0) bit-reversed addressing for accesses that are indexed with DAG2 register I8.
1	BR0	**Bit-Reverse Addressing For Index I0 Enable.** Enables (bit reversed if set, = 1) or disables (normal if cleared, = 0) bit-reversed addressing for accesses that are indexed with DAG1 register I0.
2	SRCU	**Secondary Registers For Computational Units Enable.** Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary result (MR) registers in the computational units.
3	SRD1H	**Secondary Registers For DAG1 High Enable.** Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG1 registers for the upper half (I, M, L, B7–4) of the address generator.
4	SRD1L	**Secondary Registers For DAG1 Low Enable.** Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG1 registers for the lower half (I, M, L, B3–0) of the address generator.
5	SRD2H	**Secondary Registers For DAG2 High Enable.** Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG2 registers for the upper half (I, M, L, B15–12) of the address generator.
6	SRD2L	**Secondary Registers For DAG2 Low Enable.** Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary DAG2 registers for the lower half (I, M, L, B11–8) of the address generator.
7	SRRFH	**Secondary Registers For Register File High Enable.** Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary data registers for the upper half (R15–8) of the computational units.
9–8	Reserved	
10	SRRFL	**Secondary Registers For Register File Low Enable.** Enables (use secondary if set, = 1) or disables (use primary if cleared, = 0) secondary data registers for the lower half (R7–0) of the computational units.

Table B-2. Mode Control 1 Register (MODE1) Bit Descriptions (Cont'd)

Bit	Name	Description
11	NESTM	**Nesting Multiple Interrupts Enable.** Enables (nest if set, = 1) or disables (no nesting if cleared, = 0) interrupt nesting in the interrupt controller. When interrupt nesting is disabled, a higher priority interrupt can not interrupt a lower priority interrupt's service routine. Other interrupts are latched as they occur, but the processor processes them after the active routine finishes. When interrupt nesting is enabled, a higher priority interrupt can interrupt a lower priority interrupt's service routine. Lower interrupts are latched as they occur, but the processor processes them after the nested routines finish.
12	IRPTEN	**Global Interrupt Enable.** Enables (if set, = 1) or disables (if cleared, = 0) all maskable interrupts.
13	ALUSAT	**ALU Saturation Select.** Selects whether the computational units saturate results on positive or negative fixed–point overflows (if 1) or return unsaturated results (if 0).
14	SSE	**Fixed–Point Sign Extension Select.** Selects whether the computational units sign-extend short-word, 16-bit data (if 1) or zero-fill the upper 32 bits (if 0).
15	TRUNC	**Truncation Rounding Mode Select.** Selects whether the computational units round results with round-to-zero (if 1) or round-to-nearest (if 0).
16	RND32	**Rounding For 32-Bit Floating-Point Data Select.** Selects whether the computational units round floating-point data to 32 bits (if 1) or round to 40 bits (if 0).
18–17	CSEL	**Bus Master Selection.** These bits indicate whether the processor processor has control of the external bus as follows: 00 = processor is bus master or 01, 10, 11 = processor is not bus master.
20–19	Reserved	
21	PEYEN	**Processor Element Y Enable.** Enables computations in PEy—SIMD mode—(if 1) or disables PEy—SISD mode—(if 0). When set, Processing Element Y (computation units and register files) accepts instruction dispatches. When cleared, Processing Element Y goes into a low power mode.

Table B-2. Mode Control 1 Register (MODE1) Bit Descriptions (Cont'd)

Bit	Name	Description
22	BDCST9	**Broadcast Register Loads Indexed With I9 Enable.** Enables (broadcast I9 if set, = 1) or disables (no I9 broadcast if cleared, = 0) broadcast register loads for loads that use the data address generator I9 index. When the BDCST9 bit is set, data register loads from the PM data bus that use the I9 DAG2 Index register are "broadcast" to a register or register pair in each PE.
23	BDCST1	**Broadcast Register Loads Indexed With I1 Enable.** Enables (broadcast I1 if set, = 1) or disables (no I1 broadcast if cleared, = 0) broadcast register loads for loads that use the data address generator I1 index. When the BDCST1 bit is set, data register loads from the DM data bus that use the I1 DAG1 Index register are "broadcast" to a register or register pair in each PE.
24	CBUFEN	**Circular Buffer Addressing Enable.** Enables (circular if set, = 1) or disables (linear if cleared, = 0) circular buffer addressing for buffers with loaded I, M, B, and L DAG registers.
31–25	Reserved	

Mode Mask Register (MMASK)

This is a non memory-mapped, universal, system register (*Ureg* and *Sreg*). The reset value for this register is 0x0020 0000. Each bit in the `MMASK` register corresponds to a bit in the `MODE1` register. Bits that are set in the `MMASK` register are used to clear bits in the `MODE1` register when the processor's status stack is pushed. This effectively disables different modes upon servicing an interrupt, or when executing a `PUSH STS` instruction.

The processor's status stack is pushed in two cases:

1. When executing a `PUSH STS` instruction explicitly in your code.
2. When an $\overline{\texttt{IRQ}}$`2-0` or timer expired interrupt occurs.

Example

Before the `PUSH STS` instruction, the `MODE1` register is set to 0x01202811. This `MODE1` register value corresponds to the following configuration:

- Bit-reversing for `I8`
- Secondary registers for DAG2 (high)
- Interrupt nesting, ALU saturation
- Processor element Y single-instruction multiple-data (SIMD)
- Circular buffering

The `MMASK` register is set to 0x0020 2001 indicating that you want to disable ALU saturation, SIMD, and bit reversing for `I8` after pushing the status stack. The value in the `MODE1` register after `PUSH STS` is 0x0100 0810. The other settings that were previously in the `MODE1` register remain the same. The only bits that are affected are those that are set both in the `MMASK` and in `MODE1` registers. These bits are cleared after the status stack is pushed. Figure B-3 and Figure B-4 provide bit information for the `MMASK` register.

Note also that the reset value of the `MMASK` register is 0x0020 0000. If the program does not make any changes to the `MMASK` register, the default setting automatically disables SIMD when servicing any of the hardware interrupts mentioned above, or during any push of the status stack.

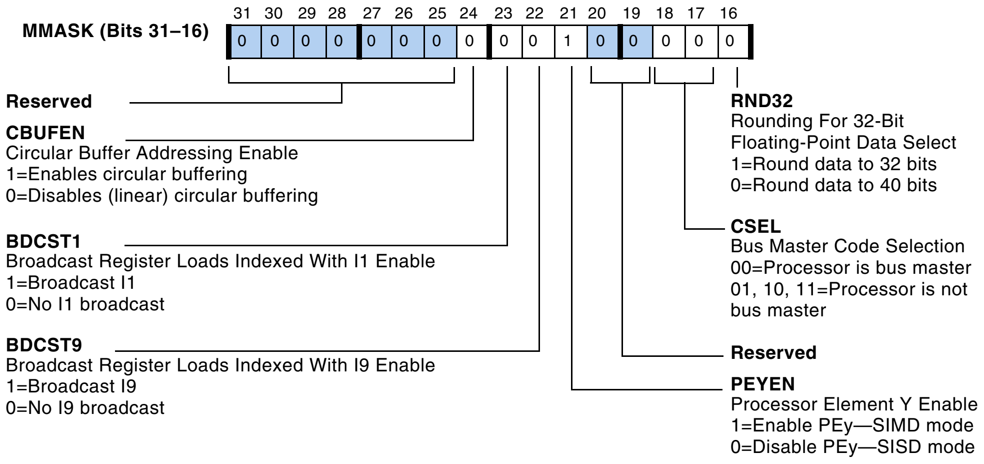

MMASK (Bits 31–16)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
Reserved
CBUFEN
Circular Buffer Addressing Enable
1=Enables circular buffering
0=Disables (linear) circular buffering
BDCST1
Broadcast Register Loads Indexed With I1 Enable
1=Broadcast I1
0=No I1 broadcast
BDCST9
Broadcast Register Loads Indexed With I9 Enable
1=Broadcast I9
0=No I9 broadcast
RND32
Rounding For 32-Bit Floating-Point Data Select
1=Round data to 32 bits
0=Round data to 40 bits
CSEL
Bus Master Code Selection
00=Processor is bus master
01, 10, 11=Processor is not bus master
Reserved
PEYEN
Processor Element Y Enable
1=Enable PEy—SIMD mode
0=Disable PEy—SISD mode

Figure B-3. MMASK Register Bits 31–16

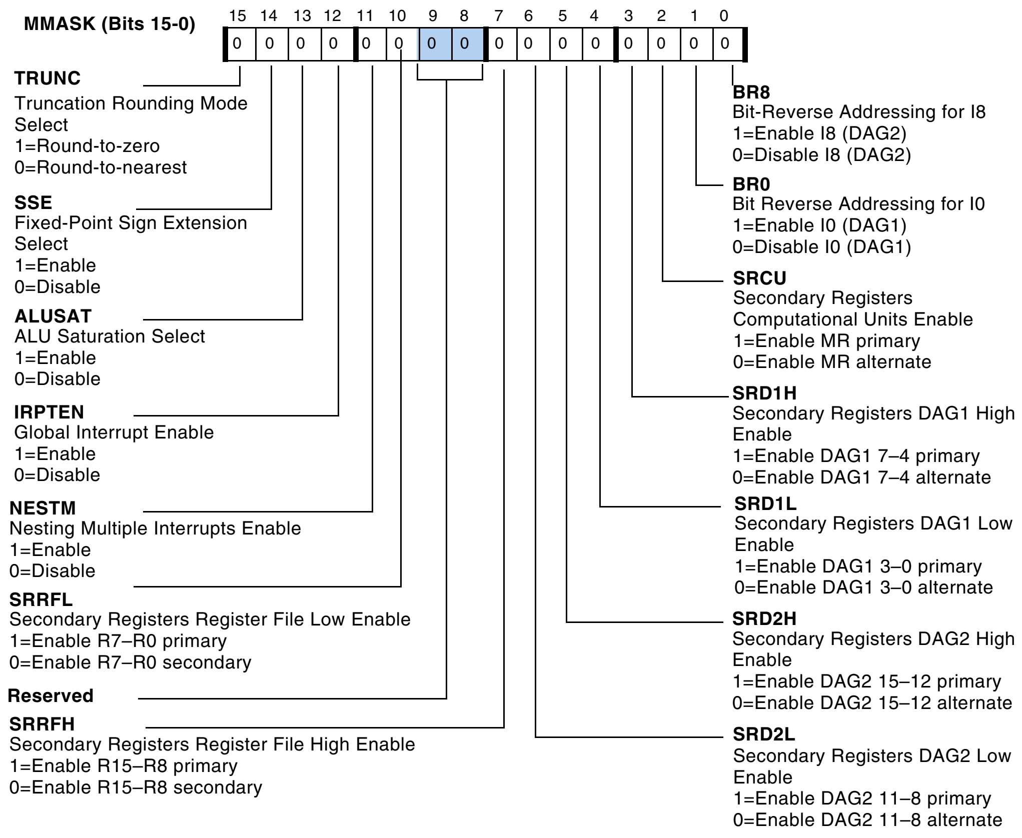

MMASK (Bits 15-0)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TRUNC
Truncation Rounding Mode Select
1=Round-to-zero
0=Round-to-nearest
SSE
Fixed-Point Sign Extension Select
1=Enable
0=Disable
ALUSAT
ALU Saturation Select
1=Enable
0=Disable
IRPTEN
Global Interrupt Enable
1=Enable
0=Disable
NESTM
Nesting Multiple Interrupts Enable
1=Enable
0=Disable
SRRFL
Secondary Registers Register File Low Enable
1=Enable R7–R0 primary
0=Enable R7–R0 secondary
Reserved
SRRFH
Secondary Registers Register File High Enable
1=Enable R15–R8 primary
0=Enable R15–R8 secondary
BR8
Bit-Reverse Addressing for I8
1=Enable I8 (DAG2)
0=Disable I8 (DAG2)
BR0
Bit Reverse Addressing for I0
1=Enable I0 (DAG1)
0=Disable I0 (DAG1)
SRCU
Secondary Registers Computational Units Enable
1=Enable MR primary
0=Enable MR alternate
SRD1H
Secondary Registers DAG1 High Enable
1=Enable DAG1 7–4 primary
0=Enable DAG1 7–4 alternate
SRD1L
Secondary Registers DAG1 Low Enable
1=Enable DAG1 3–0 primary
0=Enable DAG1 3–0 alternate
SRD2H
Secondary Registers DAG2 High Enable
1=Enable DAG2 15–12 primary
0=Enable DAG2 15–12 alternate
SRD2L
Secondary Registers DAG2 Low Enable
1=Enable DAG2 11–8 primary
0=Enable DAG2 11–8 alternate

Figure B-4. MMASK Register Bits 15–0

Mode Control 2 Register (MODE2)

The `MODE2` register is a non memory-mapped, universal, system register (*`Ureg`* and *`Sreg`*). The reset value for this register is 0x4200 0000. Figure B-5 and Table B-3 provide bit information for the `MODE2` register.

Table B-3. Mode Control 2 Register (MODE2) Bit Descriptions

Bit	Name	Description
0	$\overline{\mathrm{IRQ0}}$E	$\overline{\mathrm{IRQ0}}$ **Sensitivity Select.** Selects sensitivity for the flag configured as $\overline{\mathrm{IRQ0}}$ as edge-sensitive (if set, = 1) or level-sensitive (if cleared, = 0).
1	$\overline{\mathrm{IRQ1}}$E	$\overline{\mathrm{IRQ1}}$ **Sensitivity Select.** Selects sensitivity for the flag configured as $\overline{\mathrm{IRQ1}}$ as edge-sensitive (if set, = 1) or level-sensitive (if cleared, = 0).
2	$\overline{\mathrm{IRQ2}}$E	$\overline{\mathrm{IRQ2}}$ **Sensitivity Select.** Selects sensitivity for the flag configured as $\overline{\mathrm{IRQ2}}$ as edge-sensitive (if set, = 1) or level-sensitive (if cleared, = 0).
3	Reserved	
4	CADIS	**Cache Disable.** This bit disables the instruction cache (if set, = 1) or enables the cache (if cleared, = 0).
5	TIMEN	**Timer Enable.** Enables the timer (starts, if set, = 1) or disables the timer (stops, if cleared, = 0).
18–6	Reserved	
19	CAFRZ	**Cache Freeze.** Freezes the instruction cache (retain contents if set, = 1) or thaws the cache (allow new input if cleared, = 0).
20	IIRAE	**Illegal I/O Processor Register Access Enable.** Enables (if set, = 1) or disables (if cleared, = 0) detection of I/O processor register accesses. If IIRAE is set, the processor flags an illegal access by setting the IIRA bit in the STKYx register.
21	U64MAE	**Unaligned 64-Bit Memory Access Enable.** Enables (if set, = 1) or disables (if cleared, = 0) detection of unaligned long word accesses. If U64MAE is set, the processor flags an unaligned long word access by setting the U64MA bit in the STKYx register.
31–22	Reserved	

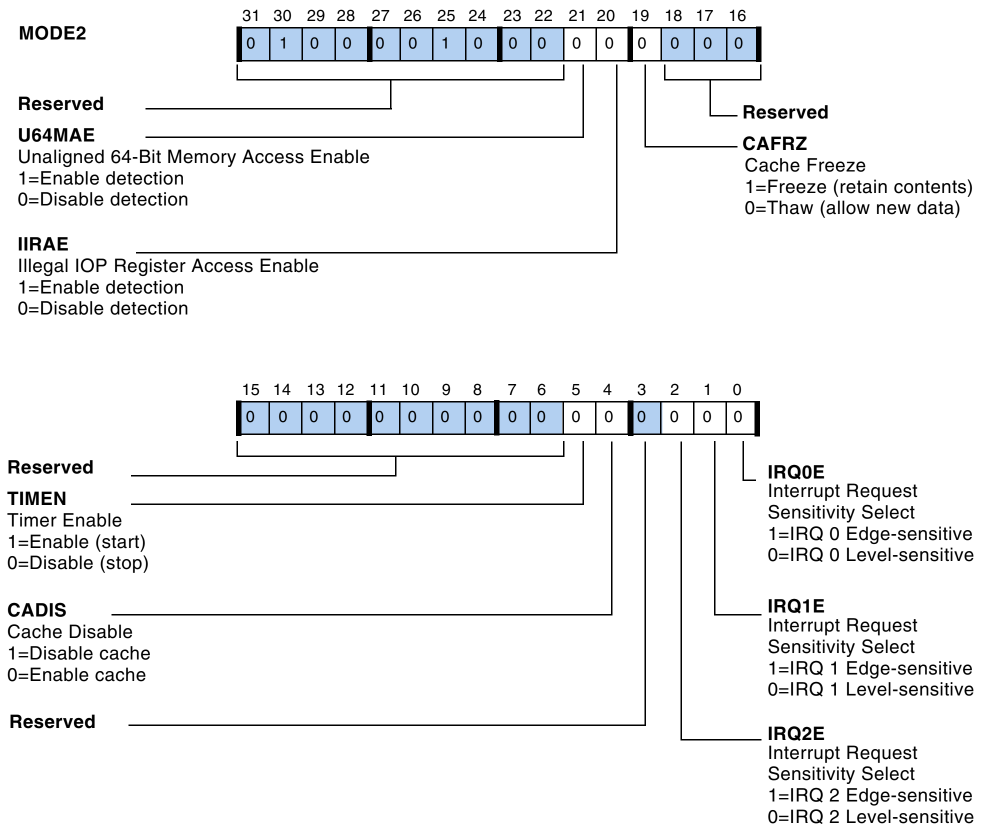


Figure B-5. MODE2 Control Register

Arithmetic Status Registers (ASTATx and ASTATy)

The ASTATx and ASTATy registers are non memory-mapped, universal, system registers (*Ureg* and *Sreg*). The reset value for these registers is 0x0000 0000. Each processing element has its own ASTAT register. The ASTATx register indicates status for PEx operations while the ASTATy register indicates status for PEy operations. Figure B-6 and Table B-4 provide bit information for the ASTAT register.

If a program loads the `ASTATx` register manually, there is a one cycle effect latency before the new value in the `ASTATx` register can be used in a conditional instruction.

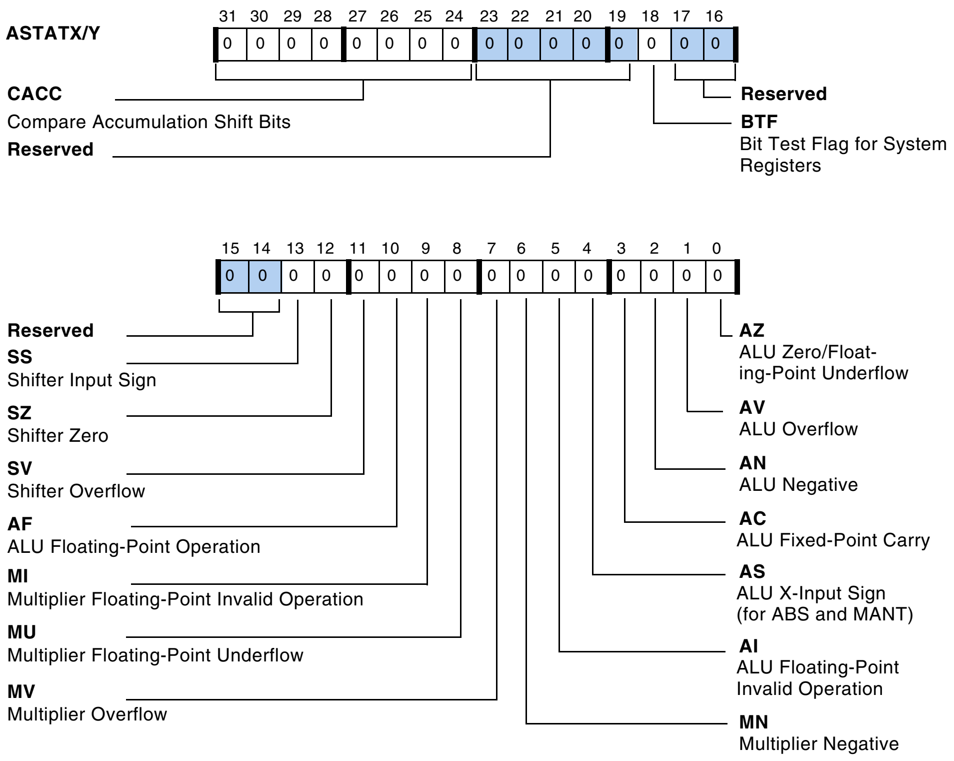


Figure B-6. ASTAT Register

Table B-4. ASTATx and ASTATy Register Bit Descriptions

Bit	Name	Description
0	AZ	**ALU Zero/Floating-Point Underflow.** Indicates if the last ALU operation's result was zero (if set, = 1) or non-zero (if cleared, = 0). The ALU updates AZ for all fixed-point and floating-point ALU operations. AZ can also indicate a floating-point underflow. During an ALU underflow (indicated by a set (= 1) AUS bit in the STKYx/y register), the processor sets AZ if the floating-point result is smaller than can be represented in the output format.
1	AV	**ALU Overflow.** Indicates if the last ALU operation's result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The ALU updates AV for all fixed-point and floating-point ALU operations. For fixed-point results, the processor sets AV and the AOS bit in the STKYx/y register when the XOR of the two most significant bits (MSBs) is a 1. For floating-point results, the processor sets AV and the AVS bit in the STKYx/y register when the rounded result overflows (unbiased exponent > 127).
2	AN	**ALU Negative.** Indicates if the last ALU operation's result was negative (if set, = 1) or positive (if cleared, = 0). The ALU updates AN for all fixed-point and floating-point ALU operations.
3	AC	**ALU Fixed-Point Carry.** Indicates if the last ALU operation had a carry out of the MSB of the result (if set, = 1) or had no carry (if cleared, = 0). The ALU updates AC for all fixed-point operations. The processor clears AC during the fixed-point logic operations: PASS, MIN, MAX, COMP, ABS, and CLIP. The ALU reads the AC flag for the fixed-point accumulate operations: Addition with Carry and Fixed-point Subtraction with Carry.
4	AS	**ALU X-Input Sign (for ABS and MANT).** Indicates if the last ALU ABS or MANT operation's input was negative (if set, = 1) or positive (if cleared, = 0). The ALU updates AS only for fixed- and floating-point ABS and MANT operations. The ALU clears AS for all operations other than ABS and MANT.

Table B-4. ASTATx and ASTATy Register Bit Descriptions (Cont'd)

Bit	Name	Description
5	AI	**ALU Floating-Point Invalid Operation.** Indicates if the last ALU operation's input was invalid (if set, = 1) or valid (if cleared, = 0). The ALU updates AI for all fixed- and floating-point ALU operations. The processor sets AI and AIS in the STKYx/y register if the ALU operation: • Receives a NAN input operand • Adds opposite-signed infinities • Subtracts like-signed infinities • Overflows during a floating-point to fixed-point conversion when saturation mode is not set • Operates on an infinity when the saturation mode is not set
6	MN	**Multiplier Negative.** Indicates if the last multiplier operation's result was negative (if set, = 1) or positive (if cleared, = 0). The multiplier updates MN for all fixed- and floating-point multiplier operations.
7	MV	**Multiplier Overflow.** Indicates if the last multiplier operation's result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The multiplier updates MV for all fixed-point and floating-point multiplier operations. For floating-point results, the processor sets MV and MVS in the STKYx/y register if the rounded result overflows (unbiased exponent > 127). For fixed-point results, the processor sets MV and the MOS bit in the STKYx/y register if the result of the multiplier operation is: • Twos-complement, fractional with the upper 17 bits of MR not all zeros or all ones • Twos-complement, integer with the upper 49 bits of MR not all zeros or all ones • Unsigned, fractional with the upper 16 bits of MR not all zeros • Unsigned, integer with the upper 48 bits of MR not all zeros If the multiplier operation directs a fixed-point result to an MR register, the processor places the overflowed portion of the result in MR1 and MR2 for an integer result or places it in MR2 only for a fractional result.

Table B-4. ASTATx and ASTATy Register Bit Descriptions (Cont'd)

Bit	Name	Description
8	MU	**Multiplier Floating-Point Underflow.** Indicates if the last multiplier operation's result underflowed (if set, = 1) or did not underflow (if cleared, = 0). The multiplier updates MU for all fixed- and floating-point multiplier operations. For floating-point results, the processor sets MU and the MUS bit in the STKYx/y register if the floating-point result underflows (unbiased exponent < –126). Denormal operands are treated as zeros, therefore they never cause underflows. For fixed-point results, the processor sets MU and the MUS bit in the STKYx/y register if the result of the multiplier operation is: • Twos-complement, fractional: with upper 48 bits all zeros or all ones, lower 32 bits not all zeros • Unsigned, fractional: with upper 48 bits all zeros, lower 32 bits not all zeros If the multiplier operation directs a fixed-point, fractional result to an MR register, the processor places the underflowed portion of the result in MR0.
9	MI	**Multiplier Floating-Point Invalid Operation.** Indicates if the last multiplier operation's input was invalid (if set, = 1) or valid (if cleared, = 0). The multiplier updates MI for floating-point multiplier operations. The processor sets MI and the MIS bit in the STKYx/y register if the ALU operation: • Receives a NAN input operand • Receives an Infinity and zero as input operands
10	AF	**ALU Floating-Point Operation.** Indicates if the last ALU operation was floating-point (if set, = 1) or fixed-point (if cleared, = 0). The ALU updates AF for all fixed-point and floating-point ALU operations.
11	SV	**Shifter Overflow.** Indicates if the last shifter operation's result overflowed (if set, = 1) or did not overflow (if cleared, = 0). The shifter updates SV for all shifter operations. The processor sets SV if the shifter operation: • Shifts the significant bits to the left of the 32-bit fixed-point field • Tests, sets, or clears a bit outside of the 32-bit fixed-point field • Extracts a field that is past or crosses the left edge of the 32-bit fixed-point field • Performs a LEFTZ or LEFTO operation that returns a result of 32
12	SZ	**Shifter Zero.** Indicates if the last shifter operation's result was zero (if set, = 1) or non-zero (if cleared, = 0). The shifter updates SZ for all shifter operations. The processor also sets SZ if the shifter operation performs a bit test on a bit outside of the 32-bit fixed-point field.

Table B-4. ASTATx and ASTATy Register Bit Descriptions (Cont'd)

Bit	Name	Description
13	SS	**Shifter Input Sign.** Indicates if the last shifter operation's input was negative (if set, = 1) or positive (if cleared, = 0). The shifter updates SS for all shifter operations.
17–14	Reserved	
18	BTF	**Bit Test Flag for System Registers.** Indicates if the system register bit is true (if set, = 1) or false (if cleared, = 0). The processor sets BTF when the bit(s) in a system register and value in the Bit Tst instruction match. The processor also sets BTF when the bit(s) in a system register and value in the Bit Xor instruction match.
23–19	Reserved	
31–24	CACC	**Compare Accumulation Shift Register.** Bit 31 of CACC indicates which operand was greater during the last ALU compare operation: X input (if set, = 1) or Y input (if cleared, = 0). The other seven bits in CACC form a right-shift register, each storing a previous compare accumulation result. With each new compare, the processor right shifts the values of CACC, storing the newest value in bit 31 and the oldest value in bit 24.

Sticky Status Registers (STKYx and STKYy)

These are non memory-mapped, universal, system registers (*Ureg* and *Sreg*). The reset value for these registers is 0x0540 0000. Each processing element has its own STKY register. The STKYx register indicates status for PEx operations and some program sequencer stacks. The STKYy register only indicates status for PEy operations.Figure B-8, Figure B-7, and Table B-5 provide bit information for both the STKYx and STKYy registers.

STKY bits do not clear themselves after the condition they flag is no longer true. They remain “sticky” until cleared by the program.

The processor sets a STKY bit in response to a condition. For example, the processor sets the AUS bit in the STKY register when an ALU underflow set AZ in the ASTAT register. The processor clears AZ if the next ALU operation does not cause an underflow. The AUS bit remains set until a program clears the STKY bit. Interrupt service routines (ISRs) must clear their interrupt's corresponding STKY bit so the processor can detect a reoccurrence of the condition. For example, an ISR for a floating-point underflow exception interrupt (FLTUI) clears the AUS bit in the STKY register near the beginning of the routine.

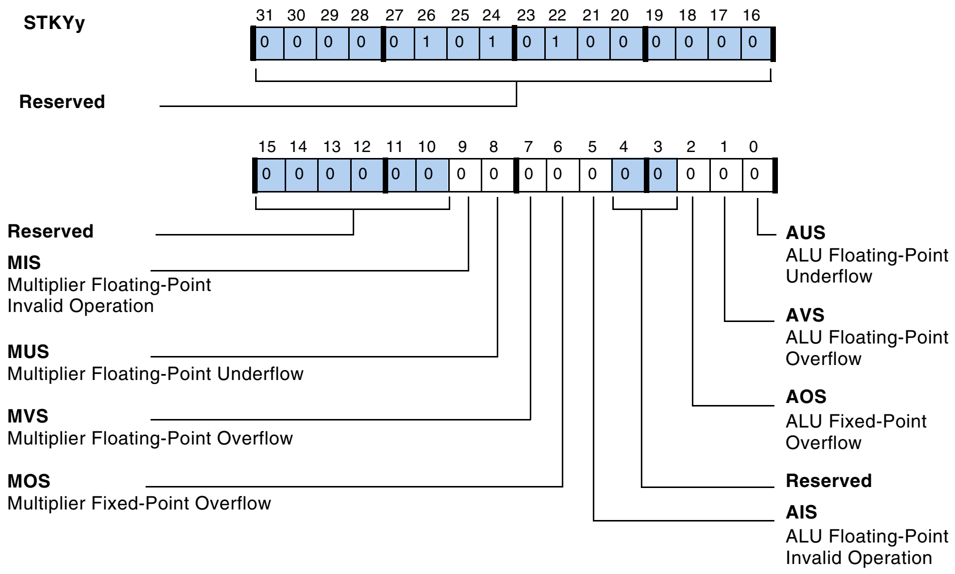

STKYy
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0
Reserved
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reserved
MIS
Multiplier Floating-Point Invalid Operation
MUS
Multiplier Floating-Point Underflow
MVS
Multiplier Floating-Point Overflow
MOS
Multiplier Fixed-Point Overflow
AUS
ALU Floating-Point Underflow
AVS
ALU Floating-Point Overflow
AOS
ALU Fixed-Point Overflow
Reserved
AIS
ALU Floating-Point Invalid Operation

Figure B-7. STKYy Register

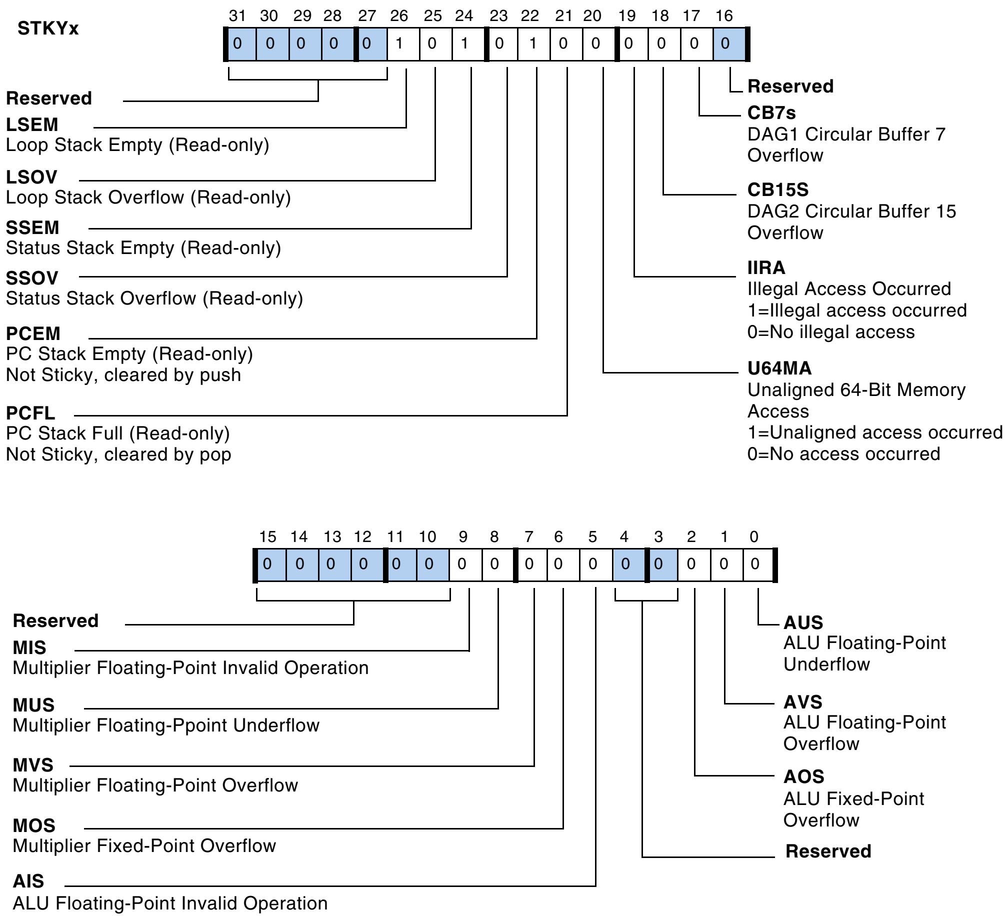

STKYx
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0
Reserved
LSEM
Loop Stack Empty (Read-only)
LSOV
Loop Stack Overflow (Read-only)
SSEM
Status Stack Empty (Read-only)
SSOV
Status Stack Overflow (Read-only)
PCEM
PC Stack Empty (Read-only)
Not Sticky, cleared by push
PCFL
PC Stack Full (Read-only)
Not Sticky, cleared by pop
Reserved
CB7s
DAG1 Circular Buffer 7 Overflow
CB15S
DAG2 Circular Buffer 15 Overflow
IIRA
Illegal Access Occurred
1=Illegal access occurred
0=No illegal access
U64MA
Unaligned 64-Bit Memory Access
1=Unaligned access occurred
0=No access occurred
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reserved
MIS
Multiplier Floating-Point Invalid Operation
MUS
Multiplier Floating-Ppoint Underflow
MVS
Multiplier Floating-Point Overflow
MOS
Multiplier Fixed-Point Overflow
AIS
ALU Floating-Point Invalid Operation
AUS
ALU Floating-Point Underflow
AVS
ALU Floating-Point Overflow
AOS
ALU Fixed-Point Overflow
Reserved

Figure B-8. STKYx Register

Table B-5. STKYx and STKYy Register Bit Descriptions

Bit	Name	Description: √ shows bits in both STKYx/y × shows bits in STKYx only	
0	AUS	**ALU Floating-Point Underflow.** A sticky indicator for the ALU AS bit. For more information, see "AZ" on page 14.	√
1	AVS	**ALU Floating-Point Overflow.** A sticky indicator for the ALU AV bit. For more information, see "AV" on page 14.	√
2	AOS	**ALU Fixed-Point Overflow.** A sticky indicator for the ALU AV bit. For more information, see "AV" on page 14.	√
4–3	Reserved		
5	AIS	**ALU Floating-Point Invalid Operation.** A sticky indicator for the ALU AI bit. For more information, see "AI" on page 15.	√
6	MOS	**Multiplier Fixed-Point Overflow.** A sticky indicator for the multiplier MV bit. For more information, see "MV" on page 15.	√
7	MVS	**Multiplier Floating-Point Overflow.** A sticky indicator for the multiplier MV bit. For more information, see "MV" on page 15.	√
8	MUS	**Multiplier Floating-Point Underflow.** A sticky indicator for the multiplier MU bit. For more information, see "MU" on page 16.	√
9	MIS	**Multiplier Floating-Point Invalid Operation.** A sticky indicator for the multiplier MI bit. For more information, see "MI" on page 16.	√
16–10	Reserved		
17	CB7S	**DAG1 Circular Buffer 7 Overflow.** Indicates if a circular buffer being addressed with DAG1 register I7 has overflowed (if set, = 1) or has not overflowed (if cleared, = 0). A circular buffer overflow occurs when DAG circular buffering operation increments the I register past the end of buffer.	×
18	CB15S	**DAG2 Circular Buffer 15 Overflow.** Indicates if a circular buffer being addressed with DAG2 register I15 has overflowed (if set, = 1) or has not overflowed (if cleared, = 0). A circular buffer overflow occurs when DAG circular buffering operation increments the I register past the end of buffer.	×

Table B-5. STKYx and STKYy Register Bit Descriptions (Cont'd)

Bit	Name	Description: √ shows bits in both STKYx/y × shows bits in STKYx only	
19	IIRA	**Illegal IOP Register Access.** Indicates if set (= 1) if a core, host, or multiprocessor access to I/O processor registers has occurred or has not occurred (if 0).	×
20	U64MA	**Unaligned 64-Bit Memory Access.** Indicates if set (= 1) if a Normal word access with the LW mnemonic addressing an uneven memory address has occurred or has not occurred (if 0).	×
21	PCFL	**PC Stack Full.** Indicates if the PC stack is full (if 1) or not full (if 0)—Not a sticky bit, cleared by a Pop.	×
22	PCEM	**PC Stack Empty.** Indicates if the PC stack is empty (if 1) or not empty (if 0)—Not sticky, cleared by a Push.	×
23	SSOV	**Status Stack Overflow.** Indicates if the status stack is overflowed (if 1) or not overflowed (if 0)—sticky bit.	×
24	SSEM	**Status Stack Empty.** Indicates if the status stack is empty (if 1) or not empty (if 0)—not sticky, cleared by a Push.	×
25	LSOV	**Loop Stack Overflow.** Indicates if the loop counter stack and loop stack are overflowed (if 1) or not overflowed (if 0)—sticky bit.	×
26	LSEM	**Loop Stack Empty.** Indicates if the loop counter stack and loop stack are empty (if 1) or not empty (if 0)—not sticky, cleared by a Push.	×
31–27	Reserved		

User-Defined Status Registers (USTATx)

These are non memory-mapped, universal, system registers (*Ureg* and *Sreg*). The reset value for these registers is 0x0000 0000. The `USTATx` registers are user-defined, general-purpose status registers. Programs can use these 32-bit registers with bit-wise instructions (`SET`, `CLEAR`, `TEST`, and others). Often, programs use these registers for low overhead, general-purpose flags or for temporary 32-bit storage of data.

Processing Element Registers

Except for the PX register, the processor's processing element registers store data for each element's ALU, multiplier, and shifter. The inputs and outputs for processing element operations go through these registers. The PX register lets programs transfer data between the data buses, but cannot be an input or output in a calculation.

Data File Data Registers (Rx, Fx, Sx)

The data file data registers are non memory-mapped, universal, data registers (*Ureg* and *Dreg*). Each of the processor's processing elements has a data register file—a set of 40-bit data registers that transfer data between the data buses and the computation units. These registers also provide local storage for operands and results.

The R, F, and S prefixes on register names do not effect the 32-bit or 40-bit data transfer; the naming convention determines how the ALU, multiplier, and shifter treat the data and determines which processing element's data registers are being used. For more information on how to use these registers, see "Data Register File" on page 2-37.

Multiplier Results Registers (MRFx, MRBx)

The MRFx and MRBx registers are non memory-mapped, universal, data registers (*Ureg* and *Dreg*). Each of the processor's multipliers has a primary or foreground (MRF) register and alternate or background (MRB) results register. As shown in Figure B-9, fixed-point operations place 80-bit results in the multiplier's foreground MRF register or background MRB register, depending on which is active. For more information on selecting the result register, see "Alternate (Secondary) Data Registers" on page 2-39. For more information on result register fields, see "Data Register File" on page 2-37.

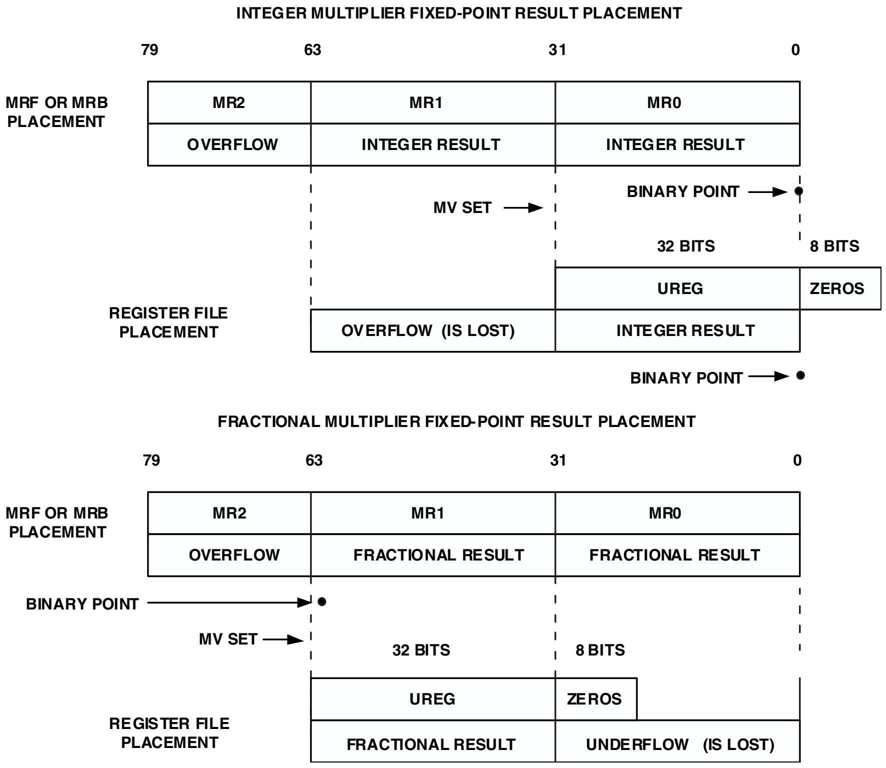


Figure B-9. MRFx and MRBx Registers

Program Memory Bus Exchange Register (PX)

The `PX` register is a non memory-mapped, universal registers (*`Ureg`* only). The PM bus exchange (`PX`) register permits data to flow between the PM and DM data buses. The `PX` register can work as one 64-bit register or as two 32-bit registers (`PX1` and `PX2`). The `PX1` register is the lower 32 bits of the `PX` register and `PX2` is the upper 32 bits of `PX`. See the section "Internal Data Bus Exchange" on page 5-7 for more information about the `PX` register.

Program Sequencer Registers

The processor's program sequencer registers, listed in Table B-6 and Table B-7, direct the execution of instructions. These registers include support for the:

- Instruction pipeline
- Program and loop stacks
- Timer
- Interrupt mask and latch

Table B-6. Program Sequencer Registers

Register	Initialization After Reset
Interrupt Latch Register (IRPTL)	0x0000 0000 (cleared)
Interrupt Mask Register (IMASK)	0x0000 0003
Interrupt Mask Pointer Register (IMASKP)	0x0000 0000 (cleared)
Interrupt Register (LIRPTL)	0x0000 0000 (cleared)
Flag Value Register (FLAGS)	0x0000 000n

Table B-7. Program Counter Registers

Register	Initialization After Reset
"Program Counter Register (PC)" on page B-30	Undefined
"Program Counter Stack Register (PCSTK)" on page B-30	Undefined
"Program Counter Stack Pointer Register (PCSTKP)" on page B-31	Undefined
"Fetch Address Register (FADDR)" on page B-31	Undefined
"Decode Address Register (DADDR)" on page B-32	Undefined
"Loop Address Stack Register (LADDR)" on page B-32	Undefined
"Current Loop Counter Register (CURLCNTR)" on page B-32	Undefined

Table B-7. Program Counter Registers (Cont'd)

Register	Initialization After Reset
"Loop Counter Register (LCNTR)" on page B-33	Undefined
"Timer Period Register (TPERIOD)" on page B-33	Undefined
"Timer Count Register (TCOUNT)" on page B-33	Undefined

Flag Value Register (FLAGS)

The FLAGS register is a non memory-mapped, universal, system register (*Ureg* and *Sreg*). At reset:

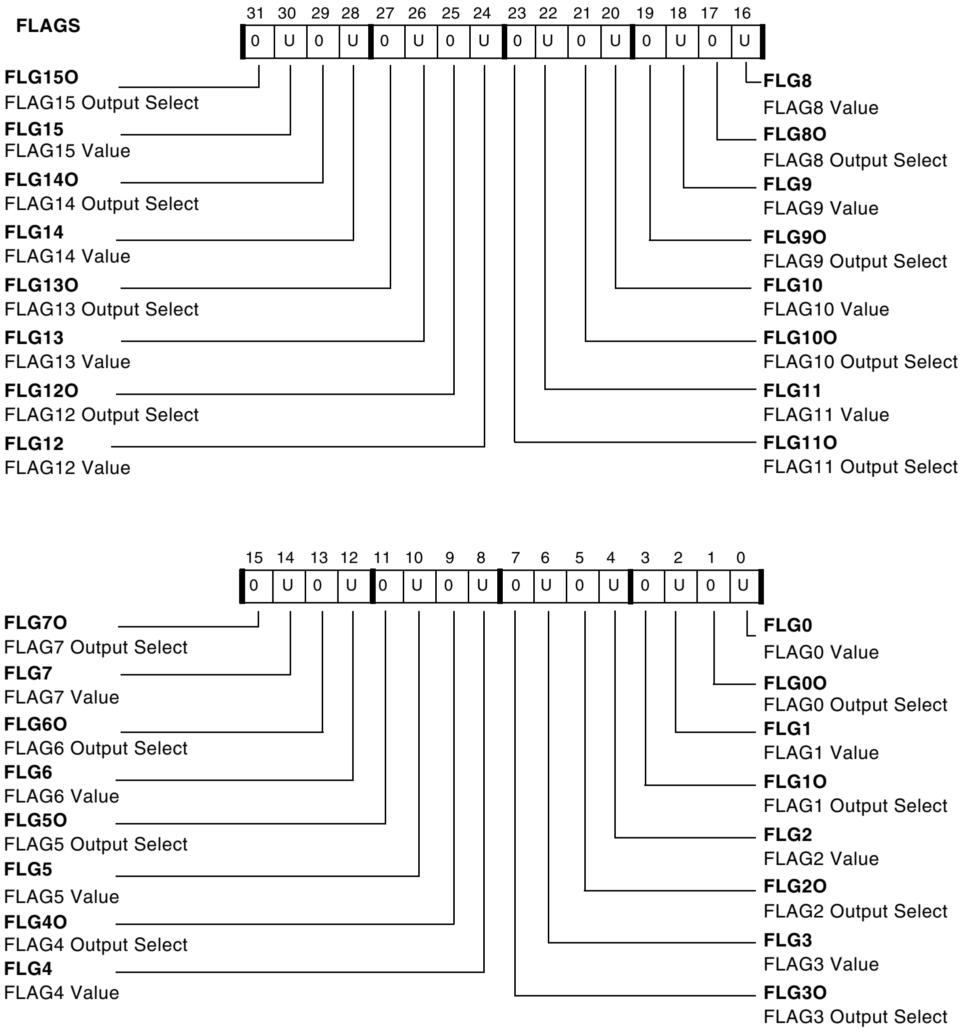
- FLG0 bit is FLAG0 pin value
- FLG1 bit is FLAG1 pin value
- FLG2 bit is FLAG2 pin value
- FLG3 bit is FLAG3 pin value
- Other FLGx bit values are unknown
- FLGx0 bits are zero

The FLAGS register indicates the state of the FLAGx pins. When a FLAGx pin is an output, the processor outputs a high in response to a program setting the bit in FLAGS. The I/O direction (input or output) selection of each bit is controlled by its FLGx0 bit in the FLAGS register. The FLAGS register bit definitions are given in Figure B-10.

There are 16 flags in ADSP-2136x processors. All are muxed with other pins. The FLAG0-3 pins have four dedicated pins. The FLAG10-15 pins are accessible to the signal routing unit (SRU). All 16 flags are routed to the AD pins when the PPFLGS bit in the SYSCTL register (= 1). While this bit is set, the parallel port is not operational and the four dedicated FLAG0-3 pins switch to their alternate state: $\overline{\text{IRQ0}}$, $\overline{\text{IRQ1}}$, $\overline{\text{IRQ2}}$, and TIMEXP.

When the SPIPDN bit (bit 30 in the PMCTL register) is set (= 1 which shuts down the clock to the SPI), the FLAGx pins cannot be used (via the FLAGS7-0 register bits) because the FLAGx pins are synchronized with the clock.

Programs cannot change the output selects of the FLAGS register and provide a new value in the same instruction. Instead, programs must use two write instructions—the first to change the output select of a particular FLAG pin, and the second to provide the new value.



-For all FLGx bits, FLAGx values are as follows: 0=LOW, 1=HIGH.
-For all FLGxO bits, FLAGx output selects are as follows: 0=FLAGx Input, 1=FLAGx Output.
-U indicates the bit value is unknown at reset.

Figure B-10. FLAGS Register

Table B-8. FLAGS Register Bit Descriptions

Bit	Name	Description
0	FLG0	**FLAG0 Value.** Indicates the state of the FLAG0 pin—high (if set, = 1) or low (if cleared, = 0).
1	FLG0O	**FLAG0 Output Select.** Selects the I/O direction for the FLAG0 pin, the flag is programmed as an output (if set, = 1) or input (if cleared, = 0).
2	FLG1	**FLAG1 Value.** Indicates the state of the FLAG1 pin—high (if set, = 1) or low (if cleared, = 0).
3	FLG1O	**FLAG1 Output Select.** Selects the I/O direction for the FLAG1 pin—an output (if set, = 1) or input (if cleared, = 0).
4	FLG2	**FLAG2 Value.** Indicates the state of the FLAG2 pin—high (if set, = 1) or low (if cleared, = 0).
5	FLAG2O	**FLAG2 Output Select.** Selects the I/O direction for the FLAG2 pin—output (if set, = 1) or input (if cleared, = 0).
6	FLAG3	**FLAG3 Value.** Indicates the state of the FLAG3 pin—high (if set, = 1) or low (if cleared, = 0).
7	FLG3O	**FLAG3 Output Select.** Selects the I/O direction for the FLAG3 pin—output (if set, = 1) or input (if cleared, = 0).
8	FLG4	**FLAG4 Value.** Indicates the state of the FLAG4 pin—high (if set, = 1) or low (if cleared, = 0).
9	FLG4O	**FLAG4 Output Select.** Selects the I/O direction for the FLAG4 pin—output (if set, = 1) or input (if cleared, = 0).
10	FLG5	**FLAG5 Value.** Indicates the state of the FLAG5 pin—high (if set, = 1) or low (if cleared, = 0).
11	FLG5O	**FLAG5 Output Select.** Selects the I/O direction for the FLAG5 pin—output (if set, = 1) or input (if cleared, = 0).
12	FLG6	**FLAG6 Value.** Indicates the state of the FLAG6 pin—high (if set, = 1) or low (if cleared, = 0).
13	FLG6O	**FLAG6 Output Select.** Selects the I/O direction for the FLAG6 pin—output (if set, = 1) or input (if cleared, = 0).
14	FLG7	**FLAG7 Value.** Indicates the state of the FLAG7 pin—high (if set, = 1) or low (if cleared, = 0).

Table B-8. FLAGS Register Bit Descriptions (Cont'd)

Bit	Name	Description
15	FLG7O	**FLAG7 Output Select.** Selects the I/O direction for the FLAG7 pin—output (if set, = 1) or input (if cleared, = 0).
16	FLG8	**FLAG8 Value.** Indicates the state of the FLAG8 pin—high (if set, = 1) or low (if cleared, = 0).
17	FLG8O	**FLAG8 Output Select.** Selects the I/O direction for FLAG8—output (if set, = 1) or an input (if cleared, = 0).
18	FLG9	**FLAG9 Value.** Indicates the state of the FLAG9 pin—high (if set, = 1) or low (if cleared, = 0).
19	FLG9O	**FLAG9 Output Select.** Selects the I/O direction for FLAG9—output (if set, = 1) or input (if cleared, = 0).
20	FLG10	**FLAG10 Value.** Indicates the state of the FLAG10 pin—high (if set, = 1) or low (if cleared, = 0).
21	FLG10O	**FLAG10 Output Select.** Selects the I/O direction for FLAG10—output (if set, = 1) or an input (if cleared, = 0).
22	FLG11	**FLAG11 Value.** Indicates the state of the FLAG11 pin—high (if set, = 1) or low (if cleared, = 0).
23	FLG11O	**FLAG11 Output Select.** Selects the I/O direction for the FLAG11—output (if set, = 1) or an input (if cleared, = 0).
24	FLG12	**FLAG12 Value.** Indicates the state of the FLAG12 pin—high (if set, = 1) or low (if cleared, = 0).
25	FLG12O	**FLAG12 Output Select.** Selects the I/O direction for FLAG12—output (if set, = 1) or input (if cleared, = 0).
26	FLG13	**FLAG13 Value.** Indicates the state of the FLAG13 pin—high (if set, = 1) or low (if cleared, = 0).
27	FLG13O	**FLAG13 Output Select.** Selects the I/O direction for FLAG13—output (if set, = 1) or an input (if cleared, = 0).
28	FLG14	**FLAG14 Value.** Indicates the state of the FLAG14 pin—high (if set, = 1) or low (if cleared, = 0).
29	FLG14O	**FLAG14 Output Select.** Selects the I/O direction for FLAG14—output (if set, = 1) or input (if cleared, = 0).

Table B-8. FLAGS Register Bit Descriptions (Cont'd)

Bit	Name	Description
30	FLG15	**FLAG15 Value.** Indicates the state of the FLAG15 pin—high (if set, = 1) or low (if cleared, = 0).
31	FLG15O	**FLAG15 Output Select.** Selects the I/O direction for FLAG15—output (if set, = 1) or input (if cleared, = 0).

Program Counter Register (PC)

The `PC` register is a non memory-mapped, universal register (*`Ureg`* only). The program counter register is the last stage in the fetch-decode-execute instruction pipeline and contains the 24-bit address of the instruction that the processor executes on the next cycle. The `PC` couples with the program counter stack, `PCSTK`, which stores return addresses and top-of-loop addresses. All addresses generated by the sequencer are 24-bit program memory instruction addresses.

As shown in Figure B-11, the address buses can handle 32-bit addresses, but the program sequencer only generates 24-bit addresses over the PM bus.

Program Counter Stack Register (PCSTK)

This is a non memory-mapped, universal register (*`Ureg`* only). The program counter stack register contains the address of the top of the `PC` stack. This is a readable and writable register.

Manipulation of these stacks by using push/pop instructions and explicit writes to these stacks may affect the correct functioning of the loop.

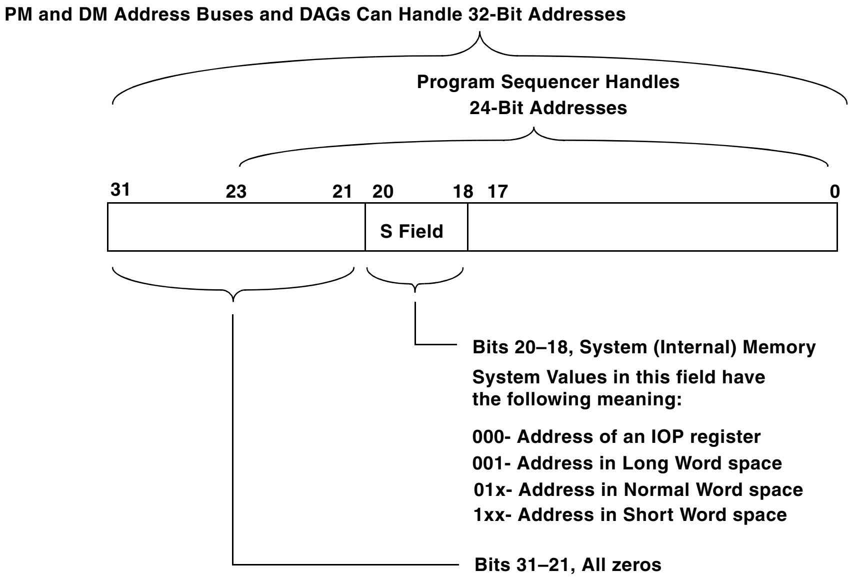

PM and DM Address Buses and DAGs Can Handle 32-Bit Addresses
Program Sequencer Handles 24-Bit Addresses
31
23
21
20
18
17
0
S Field
Bits 20–18, System (Internal) Memory
System Values in this field have the following meaning:
000- Address of an IOP register
001- Address in Long Word space
01x- Address in Normal Word space
1xx- Address in Short Word space
Bits 31–21, All zeros

Figure B-11. PM and DM Bus Addresses Versus Sequencing Addresses

Program Counter Stack Pointer Register (PCSTKP)

The `PCSTKP` register is a non memory-mapped, universal register (*`Ureg`* only). The program counter stack pointer register contains the value of `PCSTKP`. This value is given as follows: 0 when the `PC` stack is empty, 1...30 when the stack contains data, and 31 when the stack overflows. This register is readable and writable. A write to `PCSTKP` takes effect after a one-cycle delay. If the `PC` stack is overflowed, a write to `PCSTKP` has no effect.

Fetch Address Register (FADDR)

The `FADDR` register is a non memory-mapped, universal register (*`Ureg`* only). The fetch address register is the first stage in the fetch-decode-execute instruction pipeline and contains the 24-bit address of the instruction that the processor fetches from memory on the next cycle.

Decode Address Register (DADDR)

The DADDR register is a non memory-mapped, universal register (*Ureg* only). The decode address register is the second stage in the fetch-decode-execute instruction pipeline and contains the 24-bit address of the instruction that the processor decodes on the next cycle.

Loop Address Stack Register (LADDR)

The LADDR register, described in Table B-9, is a non memory-mapped, universal register (*Ureg* only). The loop address stack is six levels deep by 32 bits wide. The 32-bit word of each level consists of a 24-bit loop termination address, a 5-bit termination code, and a 2-bit loop type code.

Table B-9. LADDR Register Bit Descriptions

Bits	Value
23–0	Loop Termination Address
28–24	Termination Code
31–29	**Loop Type Code** 000 = arithmetic condition-based (not LCE) 010 = counter-based, length 1 100 = counter-based, length 2 110 = counter-based, length 3 111 = counter-based, length > 3

Current Loop Counter Register (CURLCNTR)

The CURLCNTR register is a non memory-mapped, universal register (*Ureg* only). The current loop counter register provides access to the loop counter stack and tracks iterations for the DO UNTIL LCE loop being executed. For more information on how to use the CURLCNTR register, see "Loop Status" on page 3-56.

Loop Counter Register (LCNTR)

The `LCNTR` register is a non memory-mapped, universal register (*`Ureg`* only). The loop counter register provides access to the loop counter stack and holds the count value before the `DO UNTIL LCE` loop is executed. For more information on how to use the `LCNTR` register, see "Loop Status" on page 3-56.

Timer Period Register (TPERIOD)

The `TPERIOD` register is a non memory-mapped, universal register (*`Ureg`* only). The timer period register contains the timer period, indicating the number of cycles between timer interrupts. For more information on how to use the `TPERIOD` register, see "Timer and Sequencing" on page 7-3.

Timer Count Register (TCOUNT)

The `TCOUNT` register is a non memory-mapped, universal register (*`Ureg`* only). The timer count register contains the decrementing timer count value, counting down the cycles between timer interrupts. For more information on how to use the `TCOUNT` register, see "Timer and Sequencing" on page 7-3.

Data Address Generator Registers

The processor's data address generator (DAG) registers hold data addresses, modify values, and circular buffer configurations. Using these registers, the DAGs can automatically increment addressing for ranges of data locations (a buffer).

Index Registers (Ix)

The `Ix` registers are non memory-mapped, universal registers (*`Ureg`* only). The DAGs store addresses in index registers (`I0`–`I7` for DAG1 and `I8`–`I15` for DAG2). An index register holds an address and acts as a pointer to a memory location. For more information, see "Data Address Generators" in Chapter 4, Data Address Generators.

Modify Registers (Mx)

The `Mx` register are non memory-mapped, universal registers (*`Ureg`* only). The DAGs update stored addresses using modify registers (`M0`–`M7` for DAG1 and `M8`–`M15` for DAG2). A modify register provides the increment or step size by which an index register is pre- or post-modified during a register move. For more information, see "Data Address Generators" in Chapter 4, Data Address Generators.

Length and Base Registers (Lx, Bx)

The `Lx` and `Bx` registers are non memory-mapped, universal registers (*`Ureg`* only). The DAGs control circular buffering operations with length and base registers (`L0`–`L7` and `B0`–`B7` for DAG1 and `L8`–`L15` and `B8`–`B15` for DAG2). Length and base registers set up the range of addresses and the starting address for a circular buffer. For more information, see "Data Address Generators" in Chapter 4, Data Address Generators.

Timer Registers

The timer peripheral module provides general-purpose timer functionality. It consists of three identical timer units. Each timer has four 32-bit memory-mapped registers. They are described in the following sections.

Timer Configuration Registers (TMxCTL)

All timer clocks are gated off when the specific timer's configuration register is set to zero at system reset or subsequently reset by user programs. These registers are shown in Figure B-12.

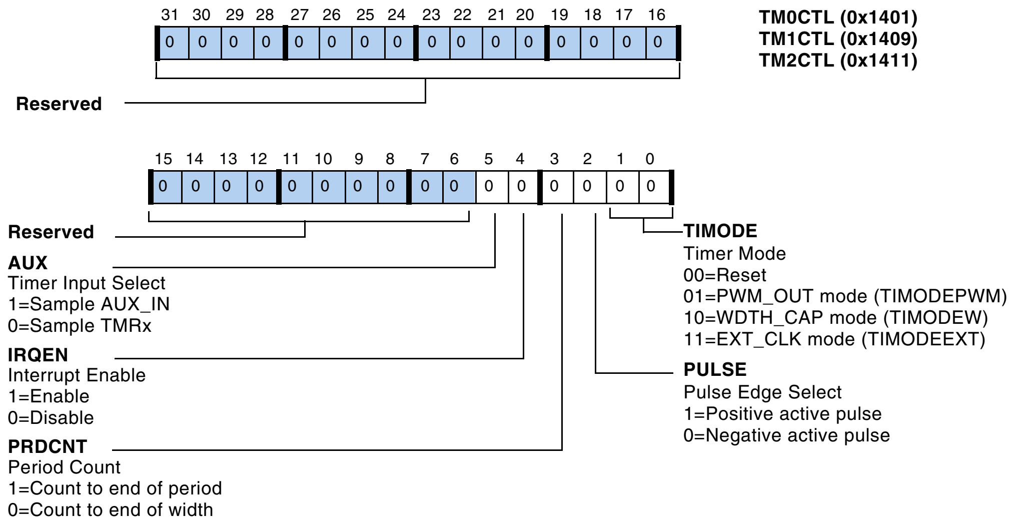


Figure B-12. Timer Configuration Register

Timer Counter Registers (TMxCNT)

The addresses for these registers are: TM0CNT = 0x1402, TM1CNT = 0x140A, and TM2CNT = 0x1412. When disabled, the timer counter retains its state. When re-enabled, the timer counter is reinitialized from the period/width registers based on configuration and mode. The timer counter value should not be set directly by the software. It can be set indirectly by initializing the period or width values in the appropriate mode. The counter should only be read when the respective timer is disabled. This prevents erroneous data from being returned.

Timer Period Registers (TMxPRD)

The addresses for these registers are: TM0PRD = 0x1403, TM1PRD = 0x140B, and TM2PRD = 0x1413. Once a timer is enabled and running, when the processor writes new values to the timer period and pulse width registers, the writes are buffered and do not update the registers until the end of the current period (when the timer counter register equals the timer period register).

During the *pulse width modulation* (PWM_OUT), the period value is written into the timer period registers. Both period and width register values must be updated "on the fly" since the period and width (duty cycle) change simultaneously. To insure the period and width value concurrency, a 32-bit period buffer and a 32-bit width buffer are used.

During the *pulse width and period capture* (WDTH_CAP) mode, the period values are captured at the appropriate time. Since both the period and width registers are read-only in this mode, the existing 32-bit period and width buffers are used.

During the *external event watchdog* (EXT_CLK) mode, the period register is write-only. Therefore, the period buffer is used in this mode to insure high/low period value coherency.

Timer Width Register (TMxW)

The addresses for these registers are: TM0W = 0x1404, TM1W = 0x140C, TM2W = 0x1414. During the pulse width modulation (PWM_OUT), the width value is written into the timer width registers. Both width and period register values must be updated "on the fly" since the period and width (duty cycle) change simultaneously. To insure period and width value concurrency, a 32-bit period buffer and a 32-bit width buffer are used.

During the pulse width and period capture (WDTH_CAP) mode, both the period and width values are captured at the appropriate time. Since both the width and period registers are read-only in this mode, the existing 32-bit period and width buffers are used.

During the EXT_CLK mode, the width register is unused.

Timer Global Status and Control Register (TMSTAT)

The global status register TMSTAT is addressable at address 0x1400 and is shown in Figure B-13. Status bits are sticky and require a write-one to clear operation. During a status register read access, all reserved or unused bits return a zero. The reset state is 0x0000. Each timer generates a unique processor interrupt request signal, TIMxIRQ.

A common status register latches these interrupts. Interrupt bits are sticky and must be cleared to assure that the interrupt is not reissued.

Each timer is provided with its own sticky status register TIMxEN bit. To enable or disable an individual timer, the TIMxEN bit is set or cleared. For example, writing a one to bit 8 sets the TIM0EN bit; writing a one to bit 9 clears it. Writing a one to both bit 8 and bit 9 clears TIM0EN. Reading the status register returns the TIM0EN state on both bit 8 and bit 9. The remaining TIMxEN bits operate similarly using bit 10 and bit 11 for timer1, and bit 12 and bit 13 for TIMER2.

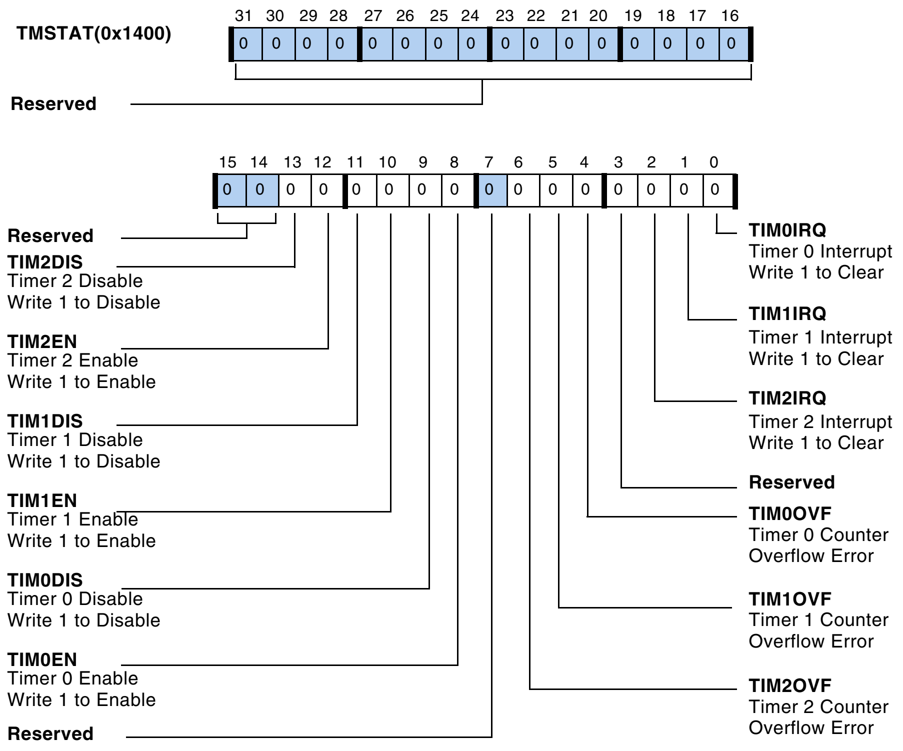


Figure B-13. TMSTAT Register

Power Management Registers

The following sections describe the registers associated with the processors power management functions.

Power Management Control Register (PMCTL)

The power management control register is a 32-bit memory-mapped register. The `PMCTL` register's addresses is 0x2000. This register contains bits to control the phase lock loop (PLL) multiplier and divider (both input and output) values, PLL bypass mode, and clock enable control for peripherals (see Figure B-14 and Table B-10). This register also contains status bits, which keep track of the status of the `CLK_CFG` pins (read-only).

The core can write to all bits except the read-only status bits. The `DIVEN` bit is a logical bit, that is, it can be set, but on reads it always responds with zero.

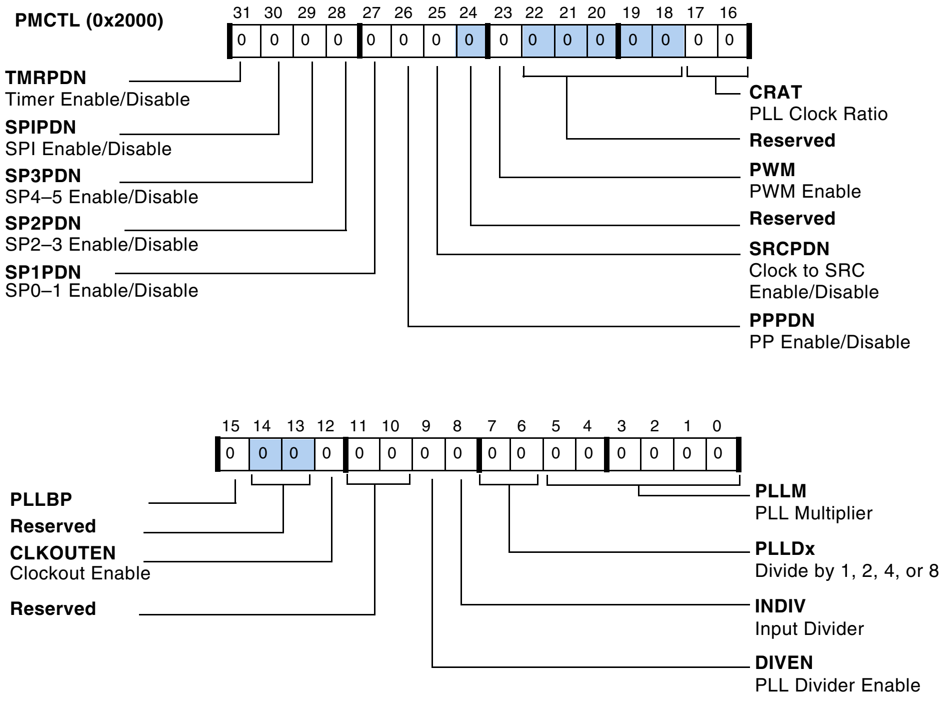


Figure B-14. PMCTL Register

Table B-10. PMCTL Register Bit Descriptions

Bit	Name	Description
5–0	PLLM	**PLL Multiplier.** Read/Write PLLM = 0 PLL Multiplier = 64 0<PLLM<63 PLL Multiplier = PLLM CLK_CFG1–0 00 = 0000110 01 = 100000 10 = 010000 11 = 000110
7–6	PLLDx	**PLL Divider.** Read/Write 00 = CK divider = 1 01 = CK divider = 2 10 = CK divider = 4 11 = CK divider = 8 CLK_CFG1–0 reset value x x 00
8	INDIV	**Input Divisor.** Read/Write 0 = Divide by 1 1 = Divide by 2 Reset value = 0
9	DIVEN	**Enable PLL Divider Value Loading.** Read/Write 0 = Do not load PLLDx 1 = Load PLLDx Reset value = 0
11–10	Reserved	
12	CLKOUTEN	**Clockout Enable.** Read/Write Mux select for CLKOUT and RESETOUT 0 Mux output = RESETOUT 1 Mux output = CLKOUT Reset value = 0
14 –13	Reserved	
15	PLLBP	**PLL Bypass Mode Indication.** Read/Write 0 = PLL is in normal mode 1 = Put PLL in bypass mode Reset value = 0

Table B-10. PMCTL Register Bit Descriptions (Cont'd)

Bit	Name	Description
17–16	CRAT	**PLL Clock Ratio (CLKIN to CK).** Read only. For more detail refer to the ADSP-2136x clock configuration pin description. Reset value = CLK_CFG1–0
22–18	Reserved	
23	PWMDN	**PWM Enable/Disable.** Shutdown clock to pulse width modulator.
24	Reserved	
25	SRCOFF	**SRC Enable/Disable.** Shutdown clock to asynchronous sample rate converter.
26	PPPDN	**PP Enable/Disable.** Read/Write 0 = Parallel port is in normal mode 1 = Shutdown clock to parallel port Reset value = 0
27	SP1PDN	**SP1 Enable/Disable.** Read/Write 0 = SP0–1 are in normal mode 1 = Shutdown clock to SP0–1 Reset value = 0
28	SP2PDN	**SP2 Enable/Disable.** Read/Write 0 = SP2–3 are in normal mode 1 = Shutdown clock to SP2–3 Reset value = 0
29	SP3PDN	**SP3 Enable/Disable.** Read/Write 0 = SP4–5 are in normal mode 1 = Shutdown clock to SP4–5 Reset value = 0

Table B-10. PMCTL Register Bit Descriptions (Cont'd)

Bit	Name	Description
30	SPIPDN	**SPI Enable/Disable.** Read/Write 0 = SPI is in normal mode 1 = Shutdown clock to SPI NOTE: When this bit is set (= 1), the FLAGx pins cannot be used (via the FLAGS7–0 register bits) because the FLAGx pins are synchronized with the clock. See "Flag Value Register (FLAGS)" on page B-25. Reset value = 0
31	TMRPDN	**Timer Enable/Disable.** Read/Write 0 = Timer is in normal mode 1 = Shutdown clock to Timer Reset value = 0

Revision ID Register (REVPID)

The `REVPID` register is top layer metal programmable 8-bit register. Because `REVPID` register bits 7–0 are the processor ID and silicon revision, the reset value varies with the system setting and silicon revision, that is, if value in top-level metal layer changes. External devices can poll this register for the processor's ID and silicon revision numbers.

As it is shown in Table B-11, the bit position from 0–3 signifies the Processor id. For ADSP-2136x processors, the process id is 0010. Bit positions 4–7 signify the silicon revision id.

Table B-11. REVPID Register Bit Descriptions

Bits	Name	Description
3–0	PID	Processor Identification (Read only) PID
7–4	Silicon Revision	Silicon Revision

I/O Processor Registers

The I/O processor's registers are accessible as part of the processor's memory map. These registers occupy addresses 0x0000 0000 through 0x0003 FFFF of the memory map. The I/O registers control the following DMA operations: parallel port, serial port, serial peripheral interface port (SPI), and input data port (IDP). The register information for the IOP and all of the peripherals associated with a specific ADSP-2136x SHARC processor is located in the processor specific *ADSP-2136x SHARC Processor Hardware Reference*. For more information, see "Related Documents" on page xxix.

G GLOSSARY

Arithmetic Logic Unit (ALU).

This part of a processing element performs arithmetic and logic operations on fixed-point and floating-point data.

Asynchronous Transfers.

Communications in which data can be transmitted intermittently rather than in a steady stream.

Auxiliary Registers.

See index registers on page G-8.

Base Address.

The starting address of a circular buffer to which the DAG wraps around. This address is stored in a DAG `Bx` register.

Base Register.

A base (`Bx`) register is a data address generator (DAG) register that sets up the starting address for a circular buffer.

Bit-Reverse Addressing.

The data address generator (DAG) provides a bit-reversed address during a data move without reversing the stored address.

Block Repeat.

See "Type 13: Do Until" on page 8-55.

Block Size Register.

See length registers on page G-9.

Boot Modes.

The boot mode determines how the processor starts up (loads its initial code). The ADSP-2136x processors can boot from its SPI port or through its parallel port via an EPROM.

Broadcast Data Moves.

The data address generator (DAG) performs dual data moves to complementary registers in each processing element to support SIMD mode.

Buffered Serial Port.

See serial ports on page G-14.

Bus Slave or Slave Mode.

A processor can be a bus slave to another processor or to a host processor. The processor becomes a host bus slave when the $\overline{\text{HBG}}$ signal is returned.

Cache Block.

The smallest unit of memory that is transferred to/from the next level of memory from/to a cache as a result of a cache miss.

Cache Hit.

A memory access that is satisfied by a valid, present entry in the cache.

Cache Line.

Same as cache block. In this document, *cache line* is used for *cache block*.

Cache Miss.

A memory access that does not match any valid entry in the cache.

Circular Buffer Addressing.

The DAG uses the `Ix`, `Mx` and `Lx` register settings to constrain addressing to a range of addresses. This range contains data that the DAG steps through repeatedly, "wrapping around" to repeat stepping through the range of addresses in a circular pattern.

Companding (Compressing/Expanding).

This is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent.

Conditional Branches.

These are `JUMP` or `CALL`/return instructions whose execution is based on testing an `IF` condition.

Core.

The core consists of these functional blocks: CPU, L1 memory, event controller, core timer, and performance monitoring registers.

Data Address Generator (DAG).

The data address generators (DAGs) provide memory addresses when data is transferred between memory and registers.

Data Register File.

This is the set of data registers that transfer data between the data buses and the computation units. These registers also provide local storage for operands and results.

Data Registers (Dreg).

These are registers in the PEx and PEy processing elements. These registers are hold operands for multiplier, ALU, or shifter operations and are denoted as `Rx` when used for fixed point operations or `Fx` when used for floating-point operations.

Deadlock Resolution.

When both the processor subsystem and the system try to access each other's bus in the same cycle, a deadlock may occur in which neither access can complete. Techniques for resolving deadlock vary with the interface: DRAM, host, or multiprocessor device.

Delayed Branches.

These are `JUMPS` and `CALL` instructions with the delayed branches `(DB)` modifier. In delayed branches, no instruction cycles are lost in the pipeline, because the processor executes the two instructions after the branch while the pipeline fills with instructions from the new branch.

Denormal Operands.

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significant zero. The numbers in this range are called denormalized (or tiny) numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented.

Direct Branches.

These are `JUMP` or `CALL` instructions that use an absolute—not changing at runtime—address (such as a program label) or use a PC-relative address.

Direct Reads and Writes.

A direct access of the processor's internal memory or I/O processor registers by another processor or by a host processor.

DMA (Direct Memory Accessing).

The processor's I/O processor supports DMA of data between processor memory and external memory, host, or peripherals through the external, link, and serial ports. Each DMA operation transfers an entire block of data.

DMA Chaining.

The processor supports chaining together multiple DMA sequences. In chained DMA, the I/O processor loads the next transfer control block (DMA parameters) into the DMA parameter registers when the current DMA finishes and auto-initializes the next DMA sequence.

DMA Parameter Registers.

These registers function similarly to data address generator registers, setting up a memory access process. These registers include internal index registers (`IISPX`, `IISPI`), internal modify registers (`IMSPI`), count registers (`CSPx`, `CSPI`), chain pointer registers (`CPSPI`), external index registers (`EIPP`), external modify registers (`EMPP`), and external count registers (`ECPP`).

DMA TCB Chain Loading.

This is the process that the I/O processor uses for loading the TCB of the next DMA sequence into the parameter registers during chained DMA.

Edge-Sensitive Interrupt.

The processor detects this type of interrupt if the input signal is high (inactive) on one cycle and low (active) on the next cycle when sampled on the rising edge of `CLKIN`.

Endian Format, Little Versus Big.

The processor uses big-endian format—moves data starting with most-significant-bit and finishing with least-significant-bit—in almost all instances. The two exceptions are bit order for data transfer through the serial port and word order for packing through the parallel port. For compatibility with little-endian (least-significant-first) peripherals, the processor supports both big- and little-endian bit order data transfers. Also for compatibility little endian hosts, the processor supports both big and little endian word order data transfers.

EPROM (Erasable Programmable Read-Only Memory).

A type of semiconductor memory in which the data is stored as electrical charges in isolated ("floating") transistor gates that retain their charges almost indefinitely without an external power supply. An EPROM is programmed by "injecting" charge into the floating gates—a process that requires relatively high voltage (usually 12V – 25V). Ultraviolet light, applied to the chip's surface through a quartz window in the package, discharges the floating gates, allowing the chip to be reprogrammed.

Explicit Versus Implicit Operations.

In SIMD mode, identical instructions execute on the PEx and PEy computational units; the difference is the data. The data registers for PEy operations are identified (implicitly) from the PEx registers in the instruction. This implicit relation between PEx and PEy data registers corresponds to complementary register pairs.

Field Deposit (Fdep) Instructions.

These shifter instructions take a group of bits from the input register (starting at the LSB of the 32-bit integer field) and deposit the bits as directed anywhere within the result register.

Field Extract (Fext) Instructions.

These shifter extract a group of bits as directed from anywhere within the input register and place them in the result register (aligned with the LSB of the 32-bit integer field).

FIFO (First In, First Out).

A hardware buffer or data structure from which items are taken out in the same order they were put in.

Flag Pins (Programmable).

These pins (`FLGx`) can be programmed as input or output pins using bit settings in the `MODE2` register. The status of the flag pins is given in the `FLAGS` or `IOFLAG` register.

Flag Update.

The processor's update to status flags occurs at the end of the cycle in which the status is generated and is available on the next cycle.

General-Purpose Input/Output Pins.

See programmable flag pins.

Harvard Architecture.

Processor's use memory architectures that have separate buses for program and data storage. The two buses let the processor get a data word and an instruction simultaneously.

Hold Time Cycle.

This is an inactive bus cycle that the processor automatically generates at the end of a read or write (depending on the parallel port access mode) to allow a longer hold time for address and data. The address—and data, if a write—remains unchanged and is driven for one cycle after the read or write strobes are deasserted.

I/O Processor Register.

One of the control, status, or data buffer registers of the processor's on-chip I/O processor.

Idle Cycle.

This is an inactive bus cycle that the processor automatically generates (depending on the parallel port access mode) to avoid data bus driver conflicts. Such a conflict can occur when a device with a long output disable time continues to drive after $\overline{\text{RD}}$ is deasserted while another device begins driving on the following cycle.

IDLE.

An instruction that causes the processor to cease operations, holding its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

Index Registers.

An index register is a data address generator (DAG) register that holds an address and acts as a pointer to memory.

Indirect Branches.

These are `JUMP` or `CALL` instructions that use a dynamic—changes at runtime—address that comes from the PM data address generator.

Inexact Flags.

An exception flag whose bit position is inexact.

Input Clock.

Device that generates a steady stream of timing signals to provide the frequency, duty cycle, and stability to allow accurate internal clock multiplication via the phase locked loop (PLL) module.

Interleaved Data.

To take advantage of the processor's data accesses to 4- and 3-column locations, programs must adjust the interleaving of data into (not necessarily sequential) memory locations to accommodate the memory access mode.

Internal Memory Space.

This space ranges from address 0x0000 0000 through 0x0005 3FFF (normal word). Internal memory space refers to the processor's on-chip SRAM and memory-mapped registers.

Interrupts.

Subroutines in which a runtime event (not an instruction) triggers the execution of the routine.

JTAG Port.

This port supports the IEEE standard 1149.1 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system.

Jumps.

Program flow transfers permanently to another part of program memory.

Latency.

The overhead time used to find the correct place for memory access and preparing to access it.

Length Registers.

A length register is a data address generator (DAG) register that sets up the range of addresses a circular buffer.

Level-Sensitive Interrupts.

The processor detects this type of interrupt if the signal input is low (active) when sampled on the rising edge of `CLKIN`.

Loops.

One sequence of instructions executes several times with zero overhead.

McBSP, Multichannel Buffered Serial Port.

See serial ports.

MCM, Multichannel Mode.

See Multichannel mode on page G-11.

Memory Access Modes.

The processor supports asynchronous external memory space. In asynchronous access mode, the processor's $\overline{\texttt{RD}}$ and $\overline{\texttt{WR}}$ strobes change before `CLKIN` edge. In synchronous access mode, the processor's $\overline{\texttt{RD}}$ and $\overline{\texttt{WR}}$ strobes change on `CLKIN` edge.

Memory Blocks and Banks.

The processor's internal memory is divided into **blocks** that are each associated with different data address generators. The processor's external memory spaces is divided into **banks**, which may be addressed by either data address generator.

Modified Addressing.

The DAG generates an address that is incremented by a value or a register.

Modify Address.

The data address generator (DAG) increments the stored address without performing a data move.

Modify Registers.

A modify register is a data address generator (DAG) register that provides the increment or step size by which an index register is pre- or post-modified during a register move.

Multichannel Mode.

In this mode, each data word of the serial bit stream occupies a separate channel.

Multifunction Computations.

Using the many parallel data paths within its computational units, the processor supports parallel execution of multiple computational instructions. These instructions complete in a single cycle, and they combine parallel operation of the multiplier and the ALU or dual ALU functions. The multiple operations perform the same as if they were in corresponding single-function computations.

Multiplier.

This part of a processing element does floating-point and fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations.

Nonzero numbers.

Nonzero, finite numbers are divided into two classes: normalized and denormalized.

Neighbor Registers.

In long word addressed accesses, the processor moves data to or from two neighboring data registers. The least-significant-32 bits moves to or from the explicit (named) register in the neighbor register pair. In forced long word accesses (normal word address with `LW` mnemonic), the processor converts the normal word address to long word, placing the even normal word location in the explicit register and the odd normal word location in the other register in the neighbor pair.

Parallel Port.

This port extends the processor's internal address and data buses off-chip, providing the processor's interface to off-chip memory devices.

PAGEN, Program Address Generation Logic.

For more information, see "Program Sequencer" on page 3-1.

Peripherals.

This refers to everything outside the processor core. The ADSP-2136x processor's peripherals include internal memory, parallel port, I/O processor, JTAG port, and any external devices that connect to the processor. Detail information about the peripherals is found in the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors* and the *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21367/8/9 Processors*.

Phase Locked Loop (PLL).

An on-chip frequency synthesizer that produces a full speed master clock from a lower frequency input clock signal.

Post-Modify Addressing.

The data address generator (DAG) provides an address during a data move and auto-increments the stored address for the next move.

Precision.

The precision of a floating-point number depends on the number of bits after the binary point in the storage format for the number. The processor supports two high precision floating-point formats: 32-bit IEEE single-precision floating-point (which uses 8 bits for the exponent and 24 bits for the mantissa) and a 40-bit extended precision version of the IEEE format.

Pre-Modify Addressing.

The data address generator (DAG) provides a modified address during a data move without incrementing the stored address.

Pulse Width Modulation.

A technique for controlling analog circuits with a microprocessor's digital outputs.

Registers Swaps.

This special type of register-to-register move instruction uses the special swap operator, <->. A register-to-register swap occurs when registers in different processing elements exchange values.

ROM (Read-Only Memory).

A data storage device manufactured with fixed contents. This term is most often used to refer to non-volatile semiconductor memory.

Saturation (ALU Saturation Mode).

In this mode, all positive fixed-point overflows return the maximum positive fixed-point number (0x7FFF FFFF), and all negative overflows return the maximum negative number (0x8000 0000).

Semaphore.

This is a flag that can be read and written by any of the processors sharing the resource. Semaphores can be used in multiprocessor systems to allow the processors to share resources such as memory or I/O. The value of the semaphore tells the processor when it can access the resource. Semaphores are also useful for synchronizing the tasks being performed by different processors in a multiprocessing system.

Serial Peripheral Interface (SPI).

A synchronous serial protocol used to connect integrated circuits.

Serial Ports (SPORTS).

The processor has six synchronous serial ports that provide an inexpensive interface to a wide variety of digital and mixed-signal peripheral devices.

SHARC.

This is an acronym for Super Harvard Architecture. This processor architecture balances a high performance processor core with high performance buses (PM, DM, I/O).

Shifter.

This part of a processing element completes logical shifts, arithmetic shifts, bit manipulation, field deposit, and field extraction operations on 32-bit operands. Also, the shifter can derive exponents.

SIMD (Single-Instruction, Multiple-Data).

A parallel computer architecture in which multiple data operands are processed simultaneously using one instruction.

SMUL, Saturation on Multiplication.

See multiplier on page G-11.

S/PDIF.

Sony/Philips Digital InterFace. A serial interface for transferring digital audio between devices such as CD and DVD players and amplifiers. S/PDIF is the consumer version of the AES/EBU interface and uses unbalanced 75 ohm coaxial cable with RCA or BNC connectors.

SP (Stack Pointer).

A register that points to the top of the stack.

SST, Saturation On Store.

See multiplier on page G-11.

Stack.

A data structure for storing items that are to be accessed in last in, first out (LIFO) order. When a data item is added to the stack, it is “pushed”; when a data item is removed from the stack, it is “popped.”

Subroutines.

The processor temporarily interrupts sequential flow to execute instructions from another part of program memory.

System Clock (SCLK).

A component that delivers clock pulses at a frequency determined by a programmable divider ratio within the PLL.

TADD, TDM Address.

See Multichannel Mode on page G-11.

TCB Chain Loading.

The process in which the processor’s DMA controller downloads a Transfer control block from memory and autoinitializes the DMA parameter registers.

Time Division Multiplexed (TDM) Mode.

The serial ports support TDM or multichannel operations. In multichannel mode, each data word of the serial bit stream occupies a separate channel— each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels.

Transfer Control Block (TCB).

A set of DMA parameter register values stored in memory that are downloaded by the processor's DMA controller for chained DMA operations.

Three-State Versus Tristate.

Analog Devices documentation uses the term "three-state" instead of "tristate" because Tristate™ is a trademarked term, which is owned by National Semiconductor.

Universal Registers (Ureg).

These are any processing element registers (data registers), any data address generator (DAG) registers, any program sequencer registers, and any I/O processor registers.

Von Neumann Architecture.

This is the architecture used by most (non-processor) microprocessors. This architecture uses a single address and data bus for memory access.

Wait States.

The time spent waiting for an operation to take place. It may refer to a variable length of time a program has to wait before it can be processed, or to a fixed duration of time, such as a machine cycle. When memory is too slow to respond to the CPU's request for it, wait states are introduced until the memory can catch up.

Write-1-to-Clear (W1C) Bit.

A control or status bit that can be cleared (= 0) by being written to with 1.

I INDEX