ADSP-2126x SHARC® DSP Core Manual

Revision 2.0, February 2004

Part Number
82-001999-01

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

Copyright Information

Printed in the USA.

Disclaimer

Trademark and Service Mark Notice

CONTENTS

PREFACE

INTRODUCTION

PROCESSING ELEMENTS

PROGRAM SEQUENCER

DATA ADDRESS GENERATORS

MEMORY

JTAG TEST EMULATION PORT

TIMER

REGISTERS

PREFACE

Thank you for purchasing and developing systems using the ADSP-2126x SHARC® DSP from Analog Devices, Inc.

Purpose of This Manual

The *ADSP-2126x SHARC DSP Core Manual* provides architectural information about the ADSP-2126x SHARC processor core. The architectural descriptions cover functional blocks, and buses, including all features and processes that they support. For programming information, see the *ADSP-21160 SHARC DSP Instruction Set Reference*. For information about the various peripherals that the ADSP-2126x DSP core can support, see the *ADSP-2126x SHARC DSP Peripherals Manual*. For timing, electrical, and package specifications, see the ADSP-21262 and ADSP-21266 data sheets.

Intended Audience

This manual is intended for ADSP-2126x processor system designers and programmers who are familiar with digital signal processing (DSP) concepts. Users should have a working knowledge of microcomputer technology and DSP related mathematics.

Manual Contents

This manual provides detailed information about the ADSP-2126x processor in the following chapters:

- "Introduction"

  Provides an architectural overview of the ADSP-2126x SHARC processor.

- "Processing Elements"

  Describes the arithmetic/logic units (ALUs), multiplier/accumulator units, and shifter. The chapter also discusses data formats, data types, and register files.

- "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.

- "Data Address Generators"

  Describes the Data Address Generators (DAGs), addressing modes, how to modify DAG and pointer registers, memory address alignment, and DAG instructions.

- "Memory"

  Describes all aspects of processor memory including internal memory, address and data bus structure, and memory accesses.

- "JTAG Test Emulation Port"

Discusses the JTAG standard and how to use the ADSP-2126x in a test environment. Includes boundary-scan architecture, instruction and boundary registers, and breakpoint control registers.

- "Timer"

  Describes the three general purpose timers that can be configured in any of three modes: pulsewidth modulation, pulsewidth count and capture, and external event watchdog modes.

Additional Literature

The following publications that describe the ADSP-2126x processor can be ordered from any Analog Devices sales office:

- *ADSP-21262 SHARC High Performance SHARC Floating-Point Processor Data Sheet*
- *ADSP-21266 SHARC High Performance SHARC Floating-Point Processor Data Sheet*
- *ADSP-21267 SHARC High Performance SHARC Floating-Point Processor Data Sheet*

What's New in This Manual

This is the second revision of the *ADSP-2126x SHARC DSP Core Manual*. The following corrections/additions have been made.

- Added section "Core Stalls" on page 3-21.
- Corrected "Circular Buffering Mode" on page 4-4. Changed to: The circular buffer enable bit (CBUFEN) in MODE1 is *cleared* (= 0) at reset.
- Figure A-21 on page A-67, changed bit 25, SRCPDN to reserved.

Technical or Customer Support

You can reach our ADSP-2126x processor Customer Support in the following ways:

- E-mail development tools questions to dsptools.support@analog.com
- E-mail processor questions to dsp.support@analog.com
- Phone questions to **1800-ANALOGD**
- Visit our World Wide Web site at http://www.analog.com/dsp
- Contact your local Analog Devices sales office or an authorized Analog Devices distributor

Processor Family

The name *ADSP-2126x* refers to the family of Analog Devices 32-bit, floating-point digital signal processors (DSP). This processor family cur-

rently includes the ADSP-21262, ADSP-21266 and ADSP-21267 SHARC processors.

Product Information

You can obtain product information from Analog Devices Web site, from the product CD-ROM, or from printed documents/manuals.

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

DSP Product Information

For information on digital signal processors, visit our Web site at http://www.analog.com/dsp. It provides access to technical information and documentation, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products by:

- FAXing questions or requests for information to **1-781-461-3010** (North America) or **089/76 903-557** (Europe Headquarters)
- Accessing the Digital Signal Processing Division FTP site: **ftp ftp.analog.com** or **ftp 137.71.23.21** or **ftp://ftp.analog.com**

Product Related Documents

For information on product related development software, see these publications:

- *VisualDSP++ User's Guide for ADSP-2126x Processors*
- *VisualDSP++ C/C++ Compiler and Library Manual for ADSP-2126x Processors*
- *VisualDSP++ Assembler and Preprocessor Manual for ADSP-2126x Processors*
- *VisualDSP++ Linker and Utilities Manual for ADSP-2126x Processors*
- *VisualDSP++ Kernel (VDK) User's Guide*
- *VisualDSP++ Component Software Engineering User's Guide*

Technical Publications Online or on the Web

You can access ADSP-2126x processor documentation in these ways:

- **Online Access using VisualDSP++® Installation CD-ROM**

  Your VisualDSP++ software distribution CD-ROM includes all of the listed VisualDSP++ software tool publications.

  After you install VisualDSP++ software on your PC, select the **Help Topics** command on the VisualDSP++ **Help** menu, click the **Reference** book icon, and select **Online Manuals**. From this **Help** topic, you can open any of the manuals, which are either in HTML format or in Adobe Acrobat PDF format.

  If you are not using VisualDSP++, you can manually access these PDF files from the CD-ROM using Adobe Acrobat.

- **Web Access**

  Use the Analog Devices technical publications Web site http://www.analog.com/industry/dsp/tech_doc/gen_purpose.html to access DSP publications, including data sheets, hardware reference books, instruction set reference books, and VisualDSP++ software documentation. You can view, download, or print in PDF format. Some publications are also available in HTML format.

Printed Manuals

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

VisualDSP++ and Tools Manuals

The VisualDSP++ and Tools manuals can be purchased through your local Analog Devices sales office or an authorized Analog Devices distributor. These manuals can only be purchased as a kit.

Hardware Manuals

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

Data Sheets

All data sheets (preliminary and production) can be downloaded from the Analog Devices Web site. As a general rule, only production (not preliminary) data sheets can be obtained from the Literature Center at

1-800-ANALOGD (**1-800-262-5643**). You can request data sheets using part numbers.

If you want to have a data sheet faxed to you, use the Analog Devices Faxback system at **1-800-446-6212**. Follow the prompts and you can either get a particular data sheet or a list of the data sheet code numbers faxed to you. If the data sheet you want is not listed on Faxback, check for it on the Web site.

Recommendations for Improving our Documents

Please send us your comments and recommendations on how to improve our manuals. Contact us at:

- Software/Development Tools manuals
  `dsptools.support@analog.com`
- Data Sheets, Hardware Reference, Programming Reference, Instruction Set Reference and User's manuals
  `dsp.support@analog.com`

Conventions

The following table identifies and describes text conventions used in this manual.

Note that additional conventions, which apply only to specific chapters, may appear throughout this document.

Table P-1. Notation Conventions

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 are delimited with a vertical bar; read the example as `this` or `that`. One or the other is required.
`{this | that}`	Optional items in syntax descriptions appear within curly braces; read the example as an optional `this` or `that`.
`[{({S|SU})}]`	Optional items for some lists may appear within parenthesis. If an option is chosen, the parenthesis must be used (for example, `(S)`). If no option is chosen, omit the parenthesis.
`.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.
`0xFBCD CBA9`	Hexadecimal numbers use the 0x prefix and are typically shown with a space between the upper four and lower four digits.
`b#1010 0101`	Binary numbers use the b# prefix and are typically shown with a space between each four digit group.
	This symbol indicates a note that provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	This symbol indicates a warning that advises on an inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Warning** appears instead of this symbol.
LSB, MSB LSW, MSW	Abbreviations for Least Significant Bit and for Most Significant Bit. Abbreviations for Least Significant Word and for Most Significant Word.

1 INTRODUCTION

Overview—Why Floating-Point DSP?

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 DSP math reduces the need for scaling and probability of overflow, using a floating-point DSP 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-2126x processor is a integrated, lower cost 32-bit floating-point DSP that provides many of these design advantages.

ADSP-2126x DSP Design Advantages

The ADSP-2126x processor is a high performance 32-bit DSP used for medical imaging, communications, military, audio, test equipment, 3D graphics, speech recognition, motor control, imaging, and other applications. By adding a dual-ported 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 DSP core to form a complete system-on-a-chip.

The SHARC processor architecture balances a high performance processor core with high performance buses (PM, DM, I/O). 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, ALU, shifter, and data register file
- Program sequencer with related instruction cache, interval timer, and Data Address Generators (DAG1 and DAG2)
- Dual-ported SRAM
- Input/Output (I/O) processor with integrated DMA controller, SPI-compatible port, and serial ports for point-to-point multiprocessor communications
- JTAG Test Access Port for emulation
- Parallel port for interfacing to off-chip memory and peripherals

Figure 1-1 also shows the three on-chip buses of the ADSP-2126x processor: the Program Memory (PM) bus, Data Memory (DM) bus, and Input/Output (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.

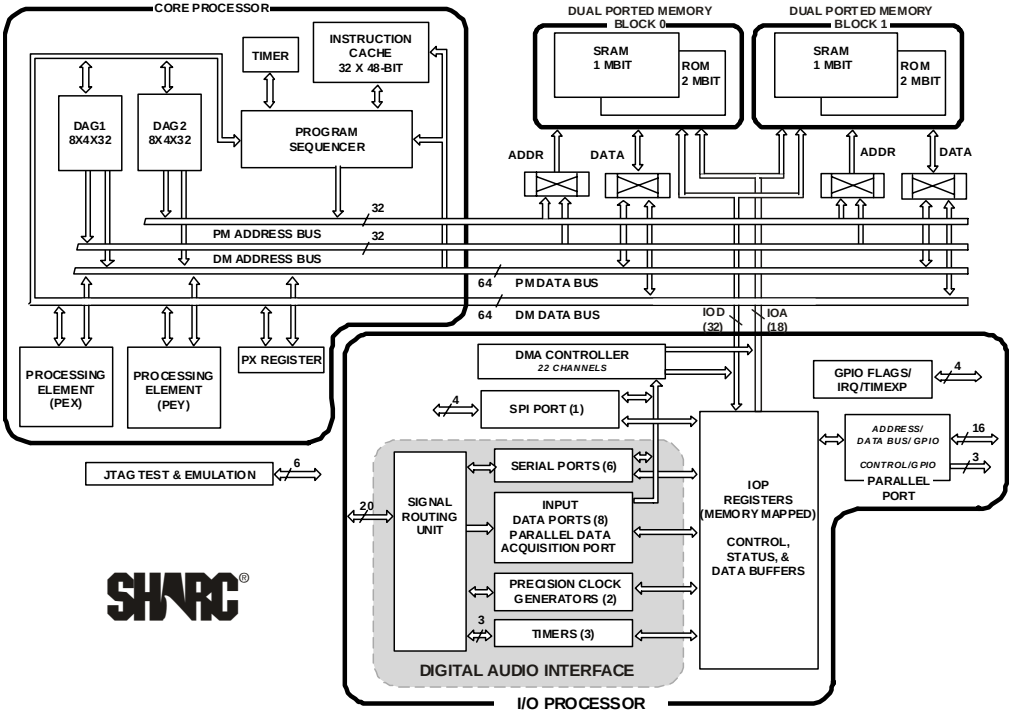


Figure 1-1. ADSP-2126x SHARC DSP Block Diagram

Further, the ADSP-2126x processor addresses the five central requirements for DSPs:

- Fast, flexible arithmetic computation units
- Unconstrained data flow to and from the computation units
- Extended precision and dynamic range in the computation units
- Dual address generators with circular buffering support
- Efficient program sequencing

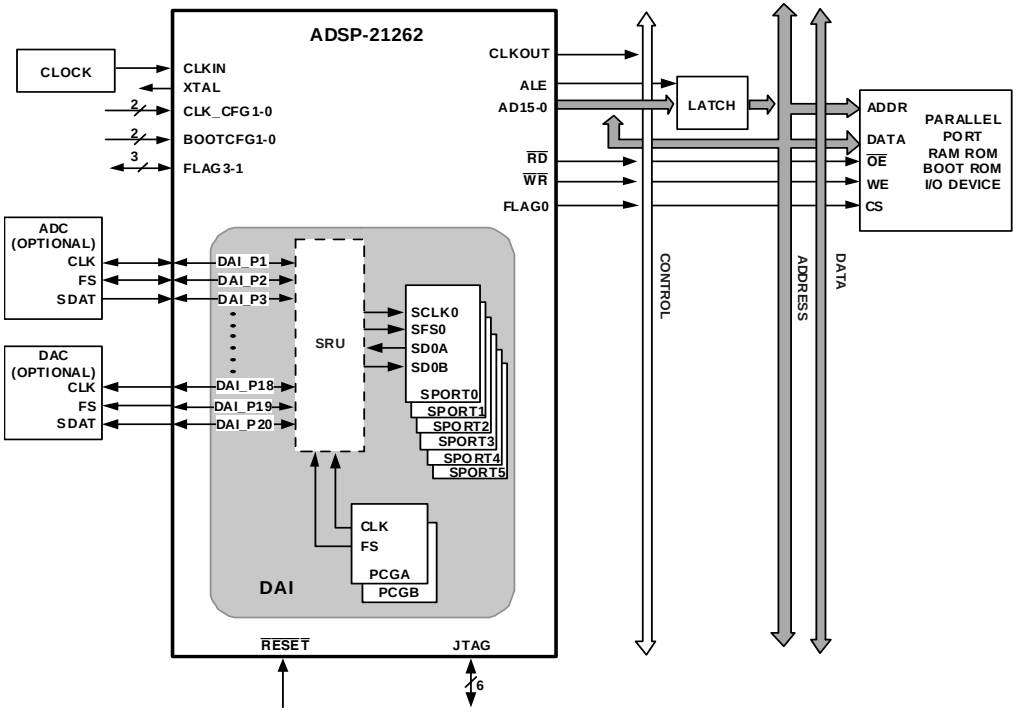


Figure 1-2. ADSP-2126x SHARC DSP Typical Single Processor System

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 DSP is IEEE floating-point compatible and allows either interrupt on arithmetic exception or latched status exception handling.

Unconstrained Data Flow. The ADSP-2126x processor has a Super Harvard Architecture combined with a ten-port data register file. In every cycle, the DSP 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.

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).

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.

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.

Architectural Overview

The ADSP-2126x processor forms 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-2126x processor architecture, which appears in Figure 1-1.

Processor Core

The processor core of the ADSP-2126x processor 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 digital signal 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 DSP is in Single-Instruction, Single-Data (SISD) or Single-Instruction, Multiple-Data (SIMD) mode. This element corre-

sponds to the computational units and register file in previous ADSP-21000 family DSPs.

Secondary Processing Element (PEy). PEy processes each computational instruction in lock-step with PEx, but only processes these instructions when the DSP 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 5-33.
- For information on SIMD programming, see the *ADSP-21160 SHARC DSP Instruction Set Reference*.

Program Sequence Control

Internal controls for ADSP-2126x processor 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-2126x processor 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 ADSP-2126x processor executes 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 DSP employs a three stage pipeline to process instructions—fetch, decode, and execute cycles.

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-2126x processor has 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 ADSP-2126x 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-2126x processor generates an interrupt and asserts its 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 mem-

ory 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 dual-ported memory.

The second port of the dual-ported memory is dedicated to the I/O address bus and the I/O data bus to let the I/O processor access internal memory for DMA without delaying the processor core. 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 A-21.

Processor Peripherals

The term processor peripherals refers to the multiple on-chip functional blocks used to communicate with off-chip devices. The ADSP-2126x processor peripherals include the JTAG, Parallel, Serial, SPI ports, DAI components (PCG, Timers, and IDP), and any external devices that con-

nect to the DSP. For complete information on using peripherals, see the peripheral user manual for the specific DSP product you are using.

Dual-Ported Internal Memory (SRAM)

The individual ADSP-2126x processor products contain varying amounts of memory. For example, the ADSP-21262 processor provides 2M bits of internal SRAM and 2M bits of internal ROM, each of which is organized as two blocks of 1M bit. Each memory block of SRAM is dual-ported for single cycle, independent accesses by the core processor and I/O processor. The dual-ported 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. The amount of memory for each word size changes, based on the part number. On the ADSP-2126x processor, the memory can be configured as a maximum of 64K words of 32-bit data, 128K words of 16-bit data, 42.5K words of 48-bit instructions (and 40-bit data), or combinations of different word sizes up to 2M bit.

The DSP 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 DSP also maintains single-cycle execution when one of the data operands is transferred to or from off chip, using the DSP parallel port.

Timers

In addition to the core's programmable interval timer, the ADSP-2126x processor has 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
- Pulsewidth 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 pulsewidth 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 on the ADSP-2126x processor 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 DSP 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 ADSP-2126x processor processors 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 DSP does not boot-load any external code, executing exclusively from internal ROM. The DSP also is not be freely accessible via the JTAG port.

Instead a 64-bit key will be 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-2126x processor is supported by VisualDSP++, an easy to use Integrated Development & 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 DSPs

This section identifies differences between the ADSP-2126x processor and previous SHARC DSPs: ADSP-21161, ADSP-21160, ADSP-21060, ADSP-21061, ADSP-21062, and ADSP-21065L. Like the ADSP-2116x family, the ADSP-2126x processor family is based on the original ADSP-2106x SHARC family. The ADSP-2126x processor 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 DSPs, see the *ADSP-2106x SHARC User's Manual* or the *ADSP-21065L SHARC Technical Reference*.

Processor Core Enhancements

Computational bandwidth on the ADSP-2126x processor is significantly greater than that on the ADSP-2106x DSPs. 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 DSP process multiple data streams in parallel (SIMD mode). The ADSP-2126x processor operates at 200 MHz using a three stage pipeline.

Like the ADSP-21160 processor, the program sequencer on the ADSP-2126x processor differs from the ADSP-2126x processor family, having 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.

As with the ADSP-21160 processor, the data address generators on the ADSP-2126x processor differ from the ADSP-2126x processor 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-2126x processor 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 on the ADSP-2126x processor have increased from 32 bits on the ADSP-2106x DSPs to 64 bits. Additional multiplexing and control logic on the ADSP-2126x processor enable 16-, 32-, or 64-bit wide moves between both register files and memory. The ADSP-2126x processor is capable of broadcasting a single memory location to each of the register files in parallel. Also, the ADSP-2126x processor permits register contents to be exchanged between the two processing elements’ register files in a single cycle.

Memory Organization Enhancements

The ADSP-2126x processor memory map differs from the memory map of the ADSP-2106x DSP. The system memory map on the ADSP-2126x processor 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-2126x processor family provides enough on-chip memory for several audio decoders.

JTAG Port Enhancements

The ADSP-2126x processor JTAG port differs from the JTAG port of the ADSP-2106x DSPs. The ADSP-2126x processor offers 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 DSP. It also gets updates from the DSP 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-2126x processor provides 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-2126x processor. Instructions, control registers, or other facilities, required to support the new feature set of the ADSP-2116x 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-2126x processor programming models

These name changes can be managed through reassembly by using the ADSP-2126x processor development tools to apply the ADSP-2126x processor 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-2126x processor has been simplified, code changes will be required to take full advantage of the new ADSP-2126x processor features. For more information, see the *ADSP-21160 SHARC DSP Instruction Set Reference*.

2 PROCESSING ELEMENTS

The DSP's processing elements (PEx and PEy) perform numeric processing for DSP 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 completes logical shifts, arithmetic shifts, bit manipulation, field deposit, and field extraction operations on 32-bit operands. Also, the shifter can 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 DSP's advanced parallelism reveal how to take advantage of multifunction instructions and Single-Instruction Multiple-Data (SIMD) mode.

Numeric Formats

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

IEEE Single-Precision Floating-Point Data Format

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).

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}.

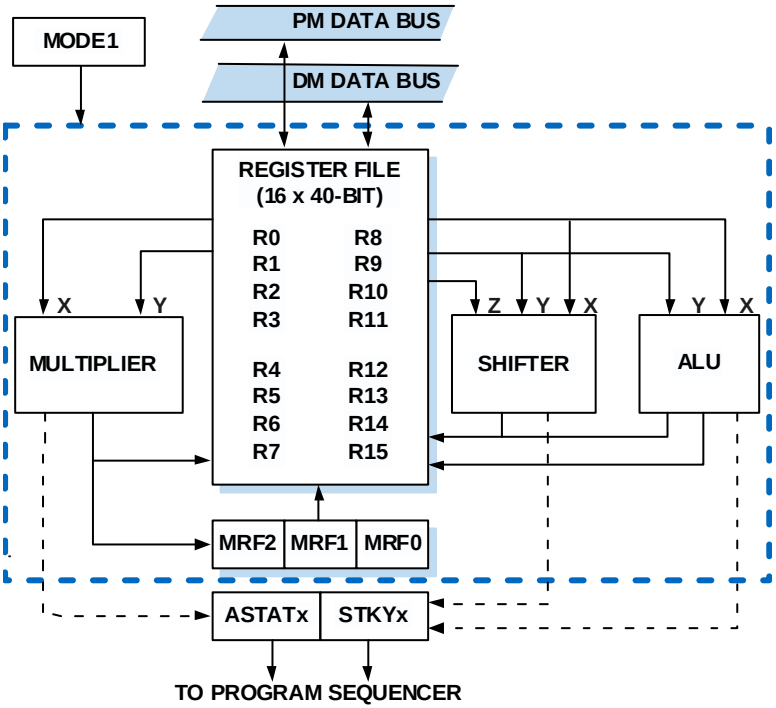


Figure 2-1. Computational Block

The Least Significant Bit (LSB) of the fraction is f_0; the LSB of the exponent is `e0`.

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 insures 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 $1 \leq e \leq 254$ for normal numbers in the single-precision format. This exponent is biased by +127 (254, 2). To calculate the true unbiased exponent, 127 must be subtracted from `e`.

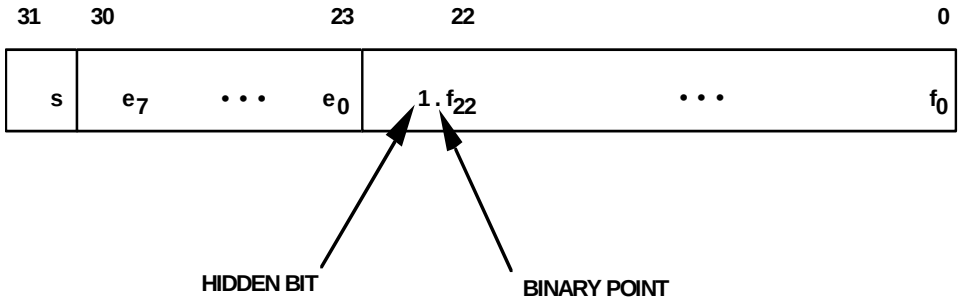


Figure 2-2. IEEE 32-Bit Single-Precision Floating-Point Format

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

- An exponent value of 255 (all ones) with a nonzero 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 * ∞.
- 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 DSP 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	Nonzero	Undefined
Infinity	255	0	(–1)σ Infinity

Table 2-1. IEEE Single-Precision Floating-Point Data Types (Cont'd)

Type	Exponent	Fraction	Value
Normal	1 ≤ e ≤ 254	Any	$(-1)^{\sigma}$ $(1.f_{22-0})$ 2^{e-127}
Zero	0	0	$(-1)^{\sigma}$ 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 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.

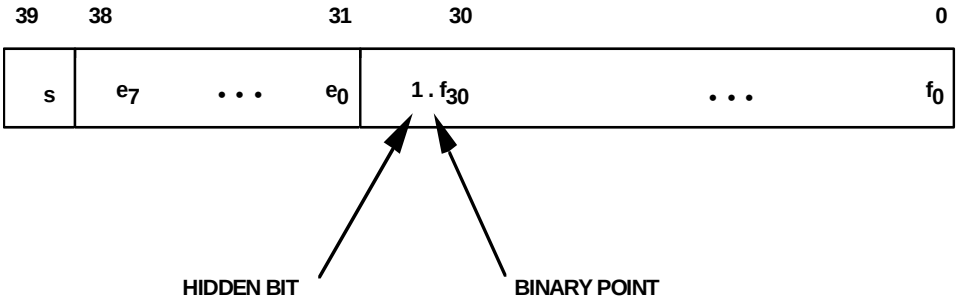


Figure 2-3. 40-Bit Extended-Precision Floating-Point Format

Short Word Floating-Point Format

The DSP 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.

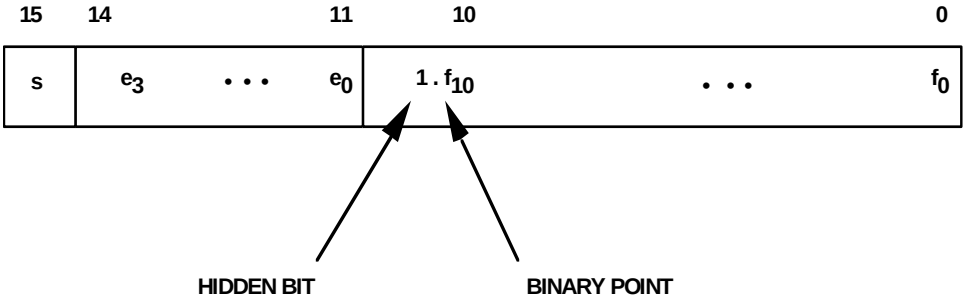


Figure 2-4. 16-Bit Floating-Point Format

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 the 16-bit floating-point numbers back to 32-bit IEEE floating-point numbers. 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.
exp = source exponent **sign bit remains the same in all cases**	

Table 2-2. FPACK Operations (Cont'd)

Condition	Result
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 DSP supports two 32-bit fixed-point formats—fractional and integer. In both formats, numbers can be signed (twos-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 twos-complement format.

ALU outputs always 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 2-7.

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 2-6.

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.

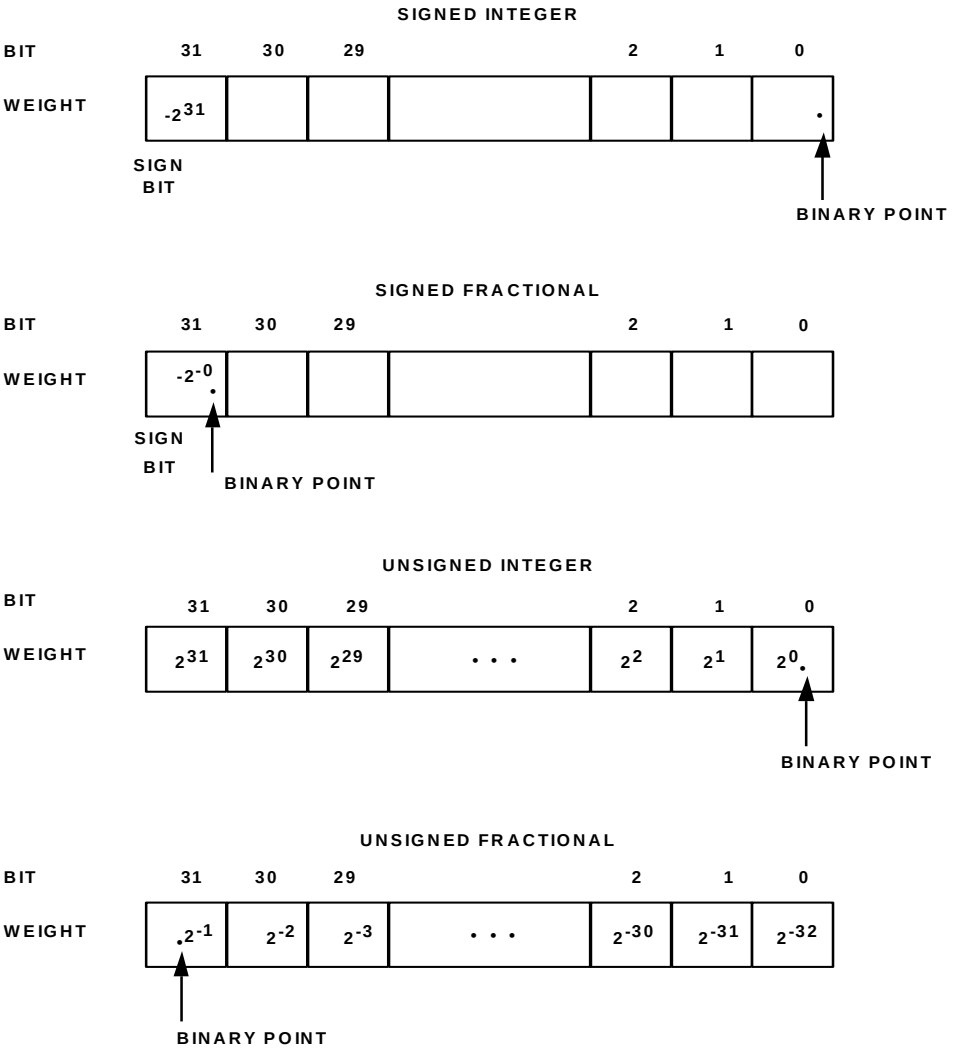


Figure 2-5. 32-Bit Fixed-Point Formats

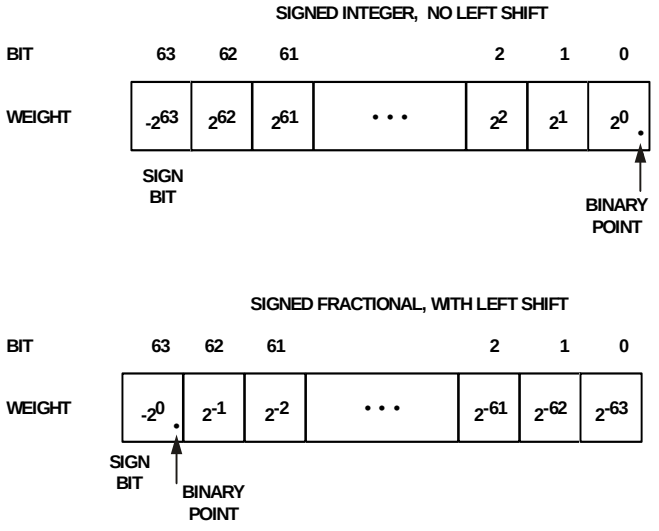


Figure 2-6. 64-Bit Signed Fixed-Point Product

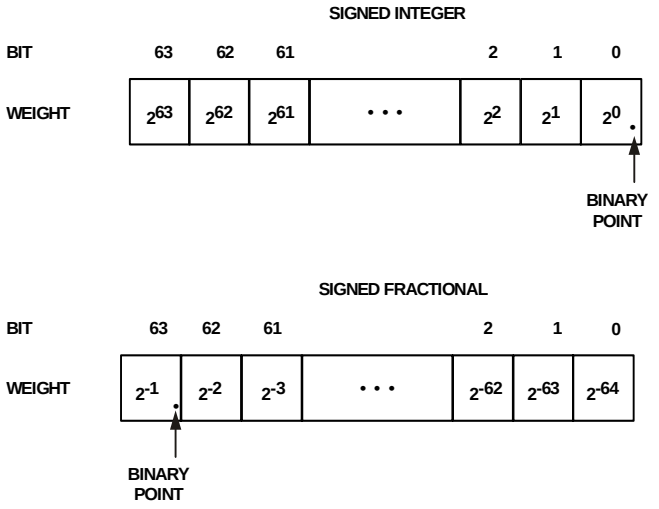


Figure 2-7. 64-Bit Unsigned Fixed-Point Product

Setting Computational Modes

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

- Floating-point data format. Bit 16 (RND32) directs the computational units to round floating-point data to 32 bits (if 1) or round to 40 bits (if 0).
- Rounding mode. Bit 15 (TRUNC) directs the computational units to round results with round-to-zero (if 1) or round-to-nearest (if 0).
- ALU saturation. Bit 13 (ALUSAT) directs the computational units to saturate results on positive or negative fixed-point overflows (if 1) or return unsaturated results (if 0).
- Short word sign extension. Bit 14 (SSE) directs the computational units to sign extended 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 Single Instruction Single Data (SISD mode) (if 0).

32-Bit Floating-Point Format (Normal Word)

In the default mode of the DSP (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 DSP 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

When in extended-precision mode (RND32 bit=0), the DSP 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 bit.

16-Bit Floating-Point Format (Short Word)

The DSP 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 DSP automatically extends the data into a 32-bit integer (bits 39 through 8). If the SSE bit in MODE1 is set (1), the DSP sign extends the upper 16 bits. If the SSE bit is cleared (0), the DSP 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 DSP always 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 twos-complement. Each computational unit has its own 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 DSP supports two modes of rounding: 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:

- Interrupts. 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.
- The ASTATx and ASTATy registers. 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.
- The STKYx and STKYy registers. 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. This method is useful 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 2-6, Table 2-7, and Table 2-8.

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, while 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 (also 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, and 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.

The DSP transfers input operands from the register file during the first half of the processor cycle and transfers results to the register file during the second half of the cycle. With this arrangement, 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 always 32-bit fixed-point values. Some floating-point operations (`Logb`, `Mant` and `Fix`) can also yield fixed-point results.

The DSP 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 twos-complement numbers.

ALU Saturation

When the `ALUSAT` bit is set (=1) in the `MODE1` register, the ALU is in 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).

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.

The ALU overflow flag reflects the ALU result before saturation.

ALU Status Flags

ALU operations update seven status flags in the processing element's arithmetic status (ASTATx and ASTATy) registers. Table A-4 on page A-14 lists all the bits in these registers. The following bits in ASTATx or ASTATy 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 A-5 on page A-19 lists all 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 the *ADSP-21160 SHARC DSP Instruction Set 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 the flag may be set or cleared, depending on the results of instruction
- ** indicates 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	–	–	–	–	**

Table 2-5. Floating-Point ALU Instruction Summary (Cont'd)

Instruction	ASTATx,y Status Flags							STKYx,y Status Flags				
Floating-point:	AZ	AV	AN	AC	AS	AI	AF	CACC	AUS	AVS	AOS	AIS
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 either 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 twos-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 result register
- Saturating result register
- Clearing result register

Multiplier Operation

The multiplier takes two inputs: X input and Y input. 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 always writes directly to the register file.

The multiplier transfers input operands during the first half of the processor cycle and transfers results during the second half of the cycle. With this arrangement, 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 either both in integer format or both in fractional format. The format of the result matches the format of the inputs. Each fixed-point operand may be either an unsigned or a twos-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.

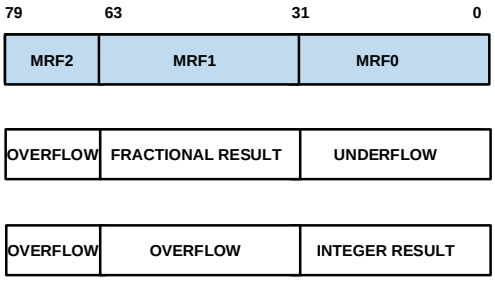


Figure 2-8. Multiplier Fixed-Point Result Placement

The `MRF` register is divided into `MRF2`, `MRF1`, and `MRF0` registers, which can be individually read from or written to the register file. Each of these registers has the same format. When data is read from `MRF2`, it is sign-extended to 32 bits as shown in Figure 2-9. The DSP zero-fills the eight LSBs of the 40-bit register file location when data is read from `MRF2`, `MRF1`, or `MRF0` to the register file. When the DSP 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.

In addition to multiplication, fixed-point operations include accumulation, rounding, and saturation of fixed-point data. There are three `MRF` register operations: clear (`Clr`), round (`Rnd`), and saturate (`Sat`).

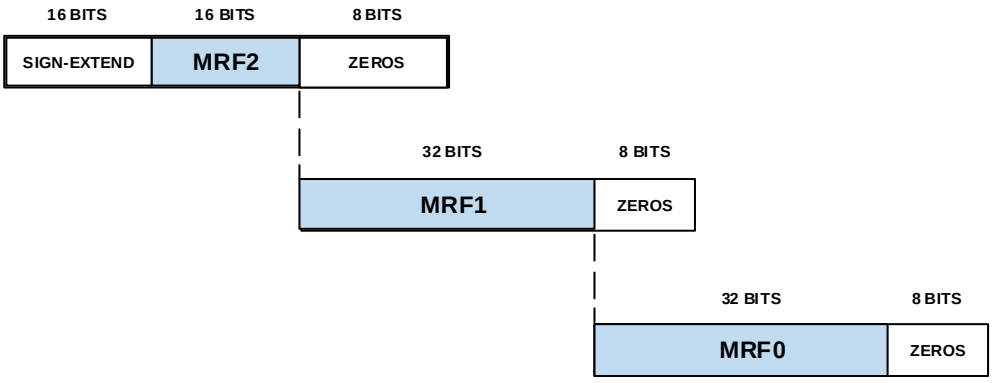


Figure 2-9. MR Transfer Formats

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 results left over from the previous operation.

The `Rnd` operation (`MRF=Rnd MRF`) applies only to fractional results, so 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 of a fixed-point result occurs either 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 either 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 twos-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 either to the register file or back to the same `MRF` register.

Table 2-6. Fixed-Point Format Maximum Values (For 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
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 A-12 lists all the bits in these registers. The following bits in the ASTATx or ASTATy registers flag the multiplier status (a 1 indicates the condition) of the most recent multiplier operation:

- 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 A-5 on page A-19 lists all the bits in these registers. The following bits in the

STKYx or STKYy flag multiplier status (a 1 indicates the condition). Once set, a sticky flag remains high until explicitly cleared:

- 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 is 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 the *ADSP-21160 SHARC DSP Instruction Set 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 the flag may be set or cleared, depending on results of instruction
- ** indicates 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	Input Mods	ASTATx,y Flags				STKYx,y Flags			
Fixed-Point: For Input Mods, see Table 2-8		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	–	**	–	–

Table 2-7. Fixed-Point Multiplier Instruction Summary (Cont'd)

Instruction	Input Mods	ASTATx,y Flags				STKYx,y Flags			
Fixed-Point: For Input Mods, see Table 2-8		MU	MN	MV	MI	MUS	MOS	MVS	MIS
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 inputS** **Unsigned inputU** **Integer inputI** **Fractional inputF** **Fractional inputs, Rounded outputFR** **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 from one to three inputs: X input, Y input, and Z input. The inputs (also 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 2-9, 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. The shifter transfers operands during the first half of the cycle and transfers the result to the upper 32 bits of a register (with the eight LSBs zero-filled) during the second half of the cycle. With this arrangement, 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 2-9.

Some shifter operations produce 8-bit or 6-bit results. As shown in Figure 2-10, the shifter places these results in either the shf8 field or the bit6 field and sign-extends the results to 32 bits. The shifter always returns a 32-bit result.

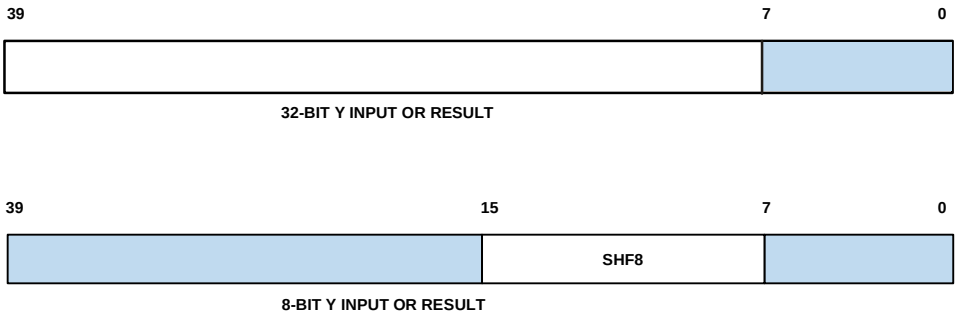


Figure 2-10. Register File Fields for Shifter Instructions

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 2-10. 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.

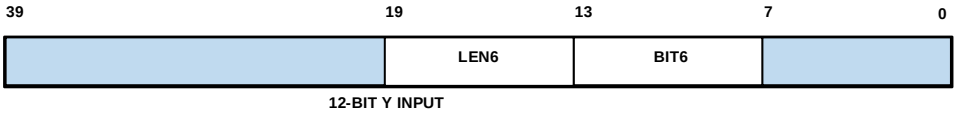


Figure 2-11. Register File Fields for FDEP, FEXT Instructions

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 2-11 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;
```

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;
```

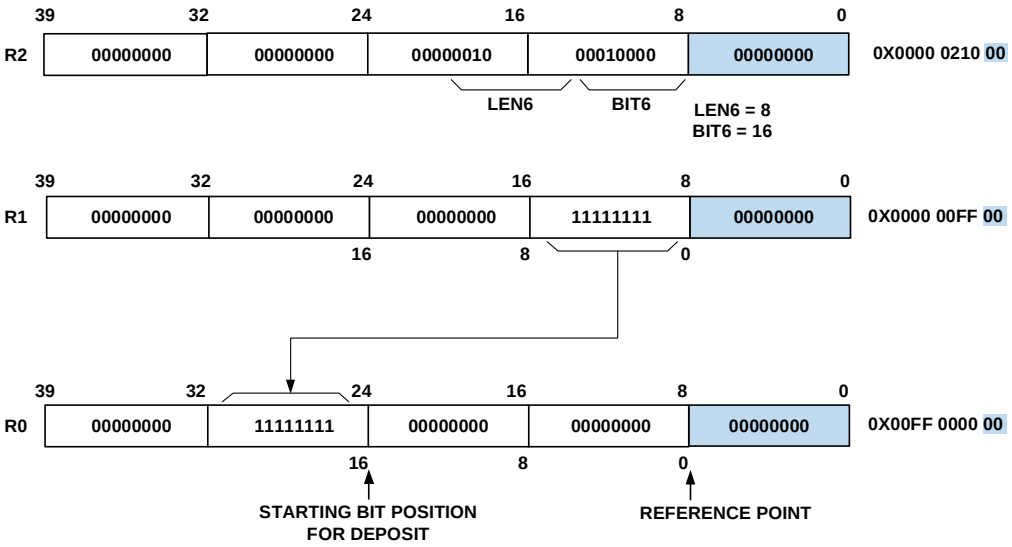


Figure 2-12. Bit Field Deposit Instruction

Shifter Status Flags

Shifter operations update three status flags in the processing element's arithmetic status registers (ASTATx and ASTATy). Table A-4 on page A-14 lists all the bits in these registers. The following bits in ASTATx or ASTATy 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)SS
- Shifter input sign for exponent extract only. Bit 13 ()

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

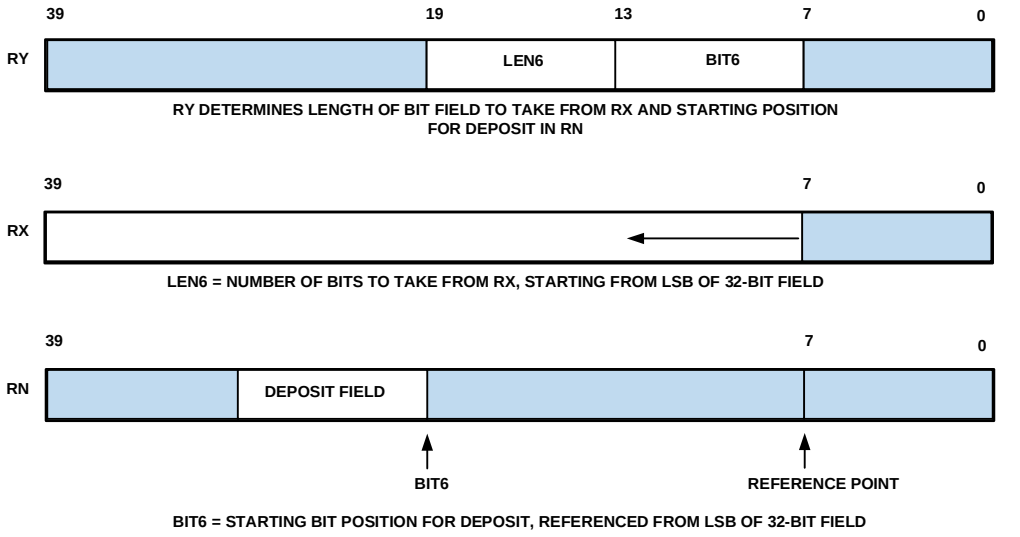


Figure 2-13. Bit Field

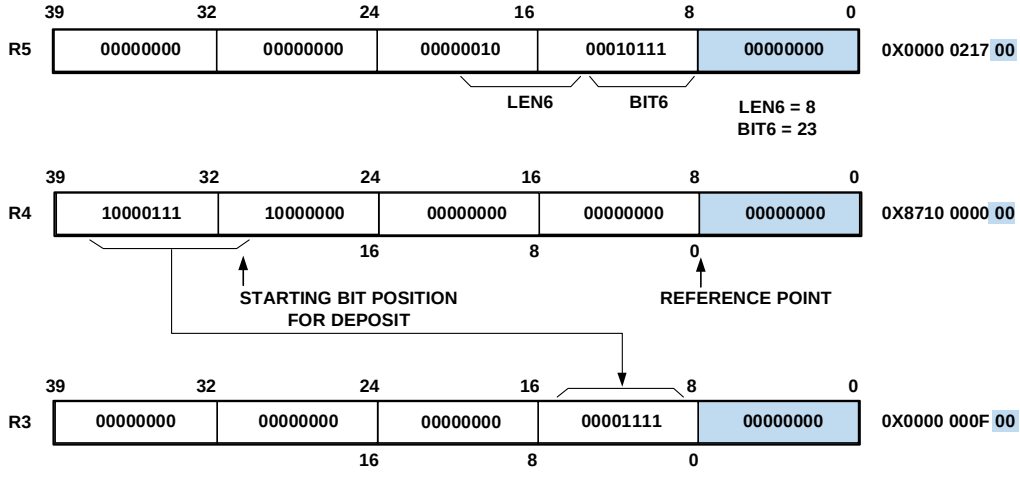


Figure 2-14. Bit Field Extract Instruction

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 the *ADSP-21160 SHARC DSP Instruction Set Reference*. In these tables, note the meaning of the following symbols:

- Rn, Rx, Ry indicate any register file location; bit fields used depend on instruction
- Fn, Fx indicate any register file location; floating-point word
- * indicates 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

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

Instruction	ASTATx,y Flags		
	SZ	SV	SS
Rn = BSET Rx BY Ry	*	*	0
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 DSP'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. All of the data registers are 40 bits wide. Within these registers, 32-bit data is always 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/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 read occurs in the first half of the cycle and the write in the second half. With this arrangement, the DSP 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 DSP 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 DSPs, the assembly syntax accommodates references to PEx data registers and PEy data registers.

Code may only refer 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 always refer to PEx registers for data move and computational instructions, whether the DSP 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 always refer to PEy registers for data move instructions, whether the DSP is in SISD or SIMD mode.

For more information on SISD and SIMD computational operations, see "Secondary Processing Element (PEy)" on page 2-44. For more information on ADSP-2126x assembly language, see the *ADSP-21160 SHARC DSP Instruction Set Reference*.

Alternate (Secondary) Data Registers

Each register file has an alternate register set. To facilitate fast context switching, the DSP 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 effected by DSP operations. Note that there is a maximum one cycle latency from the time when writes are made to MODE1 and the point when 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 the DAG "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. All fixed-point multiplies can accumulate results in either MRF or MRB, 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 A-2 on page A-5 lists all the bits in MODE1. The following bits in MODE1 control alternate registers (a 1 enables the alternate set):

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

The following example demonstrates how code should handle the maximum one cycle of latency—from the instruction that sets the bit in the MODE1 register to the point when 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 using a NOP instruction.

```
BIT SET MODE1 SRRFL;    /* activate alternate reg. file */

NOP;                    /* wait for access to alternates */

R0 = 7;
```

Multifunction Computations

The DSP supports multiple parallel (multifunction) computations by using the many 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.

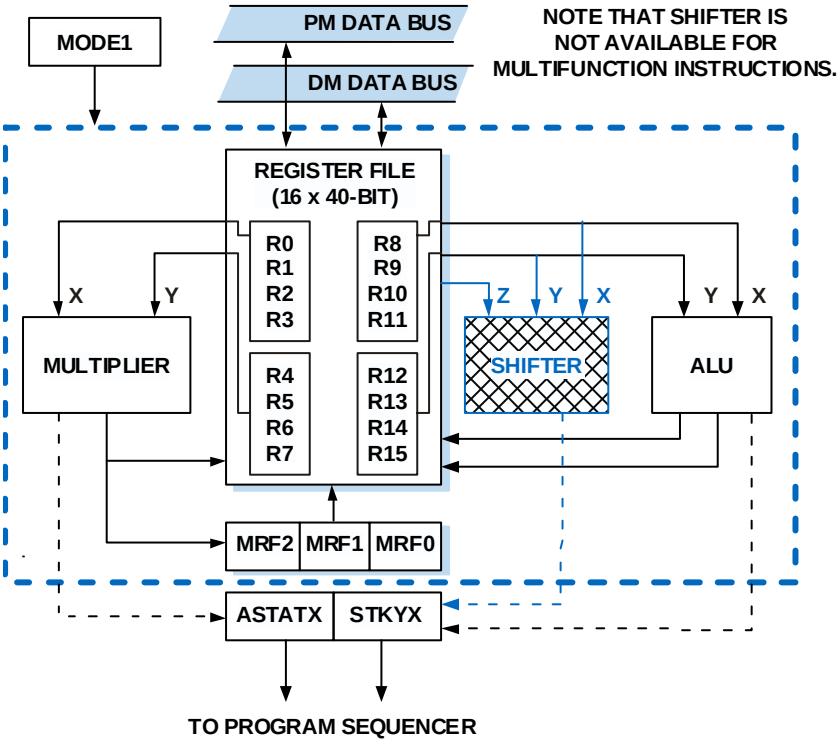


Figure 2-15. Input Registers for Multifunction Computations (ALU and Multiplier)

Table 2-11, Table 2-12, Table 2-13, and Table 2-14 list the multifunction computations. For more information on assembly language syntax, see the *ADSP-21160 SHARC DSP Instruction Set Reference*. In these tables, note the meaning of the following symbols:

- `Rm`, `Ra`, `Rs`, `Rx`, `Ry` indicate any register file location; fixed-point
- `Fm`, `Fa`, `Fs`, `Fx`, `Fy` indicate any register file location; floating-point
- `R3-0` indicates data file registers `R3`, `R2`, `R1`, or `R0`, and `F3-0` indicates data file registers `F3`, `F2`, `F1`, or `F0`

- R7-4 indicates data file registers R7, R6, R5, or R4, and F7-4 indicates data file registers F7, F6, F5, or F4
- R11-8 indicates data file registers R11, R10, R9, or R8, and F11-8 indicates data file registers F11, F10, F9, or F8
- R15-12 indicates data file registers R15, R14, R13, or R12, and F15-12 indicates data file registers F15, F14, F13, or F12
- SSFR indicates the X input is signed, the Y input is signed, use Fractional inputs, and rounded-to-nearest output
- SSF indicates the X input is signed, Y input is signed, use Fractional input

Table 2-11. Dual Add and Subtract

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

Table 2-12. 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 2-13. 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

Table 2-13. Floating-Point Multiply and ALU Operation (Cont'd)

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 2-14. 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 is also available on the DSP, combining transfers between the results and data registers and transfers between memory and data registers. As compared to other multifunction instructions, these parallel operations complete in a single cycle. For example, the DSP can perform the following multiply and parallel read of data memory:

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

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

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

Secondary Processing Element (PEy)

The ADSP-2126x 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.

The MODE1 register controls the operating mode of the processing elements. Table A-2 on page A-5 lists all the bits in MODE1. The PEYEN bit (bit 21) in the MODE1 register enables or disables the PEy processing element.

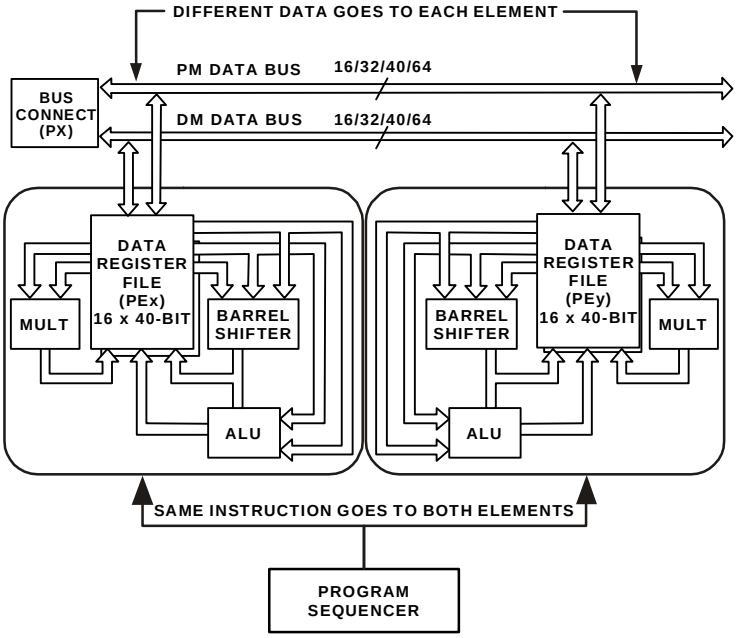


Figure 2-16. Block Diagram Showing Secondary Execution Complex

When PEYEN is cleared (0), the ADSP-2126x processor operates in SISD mode, using only PEx. When the PEYEN bit is set (1), the ADSP-2126x 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 DSP 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 the *ADSP-21160 SHARC DSP Instruction Set Reference*, it is possible through SIMD mode's parallelism to double performance over similar algorithms running in SISD (ADSP-2106x DSP 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 2-15. Any universal registers (`Ureg`) that do not appear in Table 2-15 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 2-15. SIMD Mode Complementary Register Pairs

PEx	PEy
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
ASTATx	ASTATy
STKYx	STKYy

Table 2-16. Other Complementary Register Pairs

USTAT1	USTAT2
USTAT3	USTAT4
PX1	PX2
MRF	MSF[1]
MRB	MSB1

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 Chapter 7 in *ADSP-21160 SHARC DSP Instruction Set 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— Is 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 "Summary" on page 3-63.

For information on SIMD issues relating to program sequencing, see "Addressing in SISD and SIMD Modes" on page 4-20.

Dual Alternate Registers

Both register files consist of a primary set of 16 by 40-bit registers and an alternate set of 16 by 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 DSP 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 both SISD (unidirectional) and SIMD (bidirectional) modes.

In SISD mode (PEYEN bit=0), the register-to-register transfers are unidirectional, meaning that 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 2-17 on page 2-52. Note that in SISD mode a condition for an instruction only tests in the PEx element but it applies to the entire instruction.

In SIMD mode (PEYEN bit=1), the register-to-register transfers are bidirectional, meaning that 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 2-17. Note that in SIMD mode conditional explicit and implicit transfers are tested and executed separately in PEx and PEy, respectively, as detailed in Table 2-17.

Bidirectional register-to-register transfers in SIMD mode are allowed between a data register and DAG, control, or status registers. 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.

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

In both SISD and SIMD modes, the DSP supports bidirectional register-to-register swaps. The swap always 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 2-17.

Table 2-17. 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 DSP 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 either or both processing elements generates an exception interrupt. Interrupt service routines (ISRs) must determine which of the processing elements encountered the exception. Note that returning from a

floating-point interrupt does not automatically clear the STKY state. Code must clear the STKY bits in both processing element's sticky status (STKYx and STKYy) registers as part of the exception service routine. "Interrupts and Sequencing" on page 3-46.

3 PROGRAM SEQUENCER

The DSP's program sequencer controls program flow by constantly providing the address of the next instruction to be fetched for execution. Program flow in the DSP 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 shown below. Nonsequential structures direct the DSP 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.** The traditional `CALL`/`RETURN` 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, made up of the fetch address register, decode address 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 program memory instruction addresses.

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

- "Instruction Pipeline" on page 3-2
- "Instruction Cache" on page 3-5
- "Branches and Sequencing" on page 3-11
- "Loop and Status Stacks and Sequencing" on page 3-17
- "Conditional Sequencing" on page 3-18
- "Loops and Sequencing" on page 3-23
- "SIMD Mode and Sequencing" on page 3-35
- "Timer and Sequencing" on page 3-44
- "Interrupts and Sequencing" on page 3-46

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

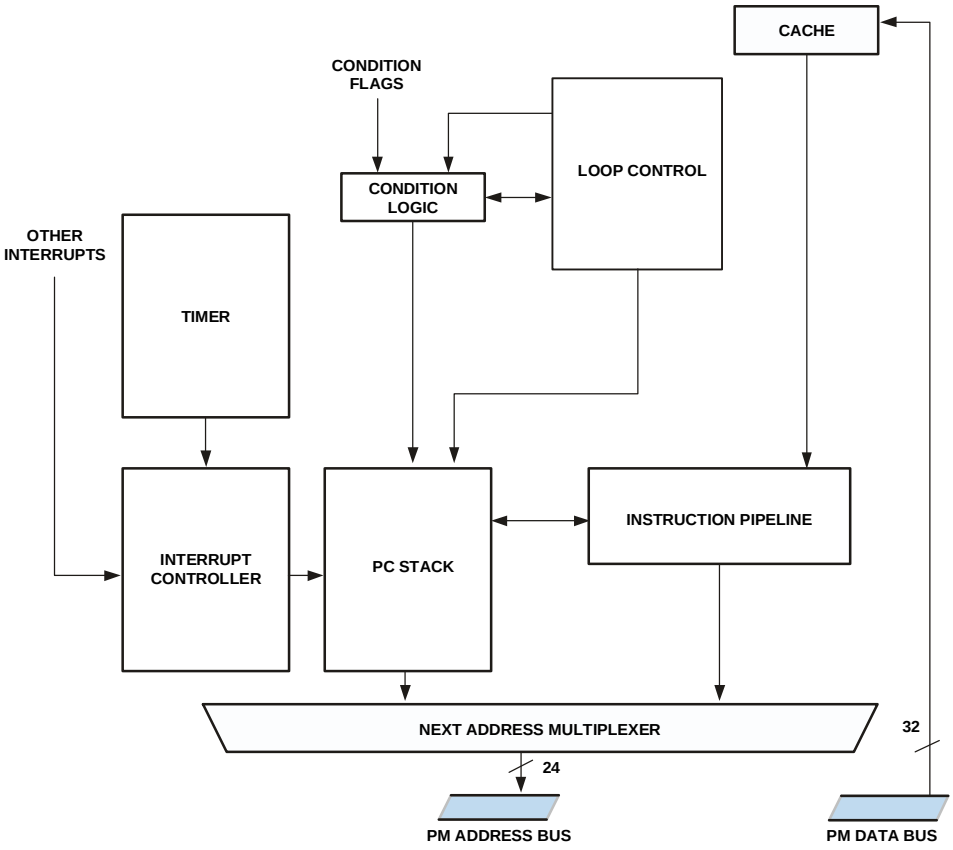


Figure 3-1. Program Sequencer Block Diagram

the processor. If no conditions require otherwise, the DSP fetches and executes instructions from memory in sequential order.

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

1. **Fetch cycle.** The DSP reads the instruction from either the on-chip memory or the instruction cache.
2. **Decode cycle.** The DSP decodes the instruction, generating conditions that control instruction execution and program flow.
3. **Execute cycle.** The DSP executes the instruction; the operations specified by the instruction complete in a single cycle.

In a sequential program flow, when one instruction is being executed, the next instruction is being decoded, and the instruction following that is being fetched. Sequential program flow usually has a throughput of one instruction per cycle. In the event of cache misses, instructions may take more than one cycle.

Figure 3-2 illustrates how the instructions starting at address 0x08 are processed by the pipeline. While the instruction at address 0x08 is being executed, the instruction 0x09 is being decoded and the instruction at address 0xA is being fetched.

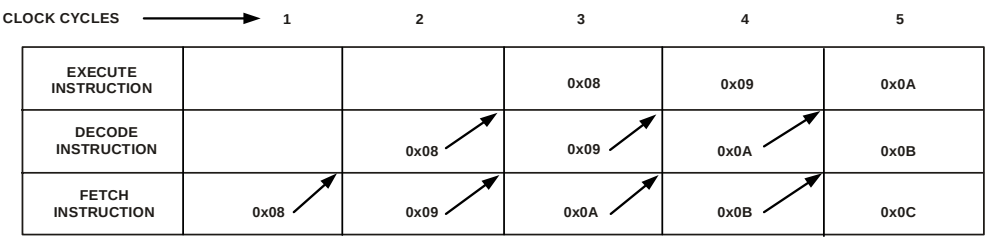


Figure 3-2. Pipelined Execution Cycles

While sequential execution takes one core clock cycle per instruction, branching (nonsequential executions) can temporarily reduce this rate. Nonsequential program operations include:

- Program memory data accesses that conflict with instruction fetches
- Jumps
- Subroutine calls and returns
- Interrupts and returns
- Loops

Instruction Cache

Usually, the sequencer fetches an instruction from memory on each cycle. Occasionally, bus constraints prevent some of the data and instructions from being fetched in a single cycle. To alleviate these data flow conflicts, the DSP has a large 32-location instruction cache that caches instructions that cause these conflicts. This solution removes the need to fetch the offending instruction from memory, which 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.

Bus Conflicts

A bus is comprised of two parts: the address bus and the data bus. Because the bus can be accessed continually by different sources (illustrated in Figure 3-1 on page 3-3), there is a potential for bus or block *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 three stage instruction pipeline, as the DSP executes an instruction (at address *n*) it also uses the PM bus to access data. For sequential executions, this creates a conflict with the instruction fetch (at address *n+2*).

The cache stores the fetched instruction (*n+2*), not the instruction requiring the program memory data access.

Block conflicts differ from bus conflicts in that block conflicts occur when there are multiple outstanding writes to the same memory block or to the same word in a different block. When the DSP first encounters a bus conflict, it must stall for one cycle while the data is transferred, and then fetch the instruction on the following cycle. To prevent the same delay from happening again, the DSP 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. This 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 3-3 shows a block diagram of the instruction cache. The cache holds 32 instuction-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 if 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.

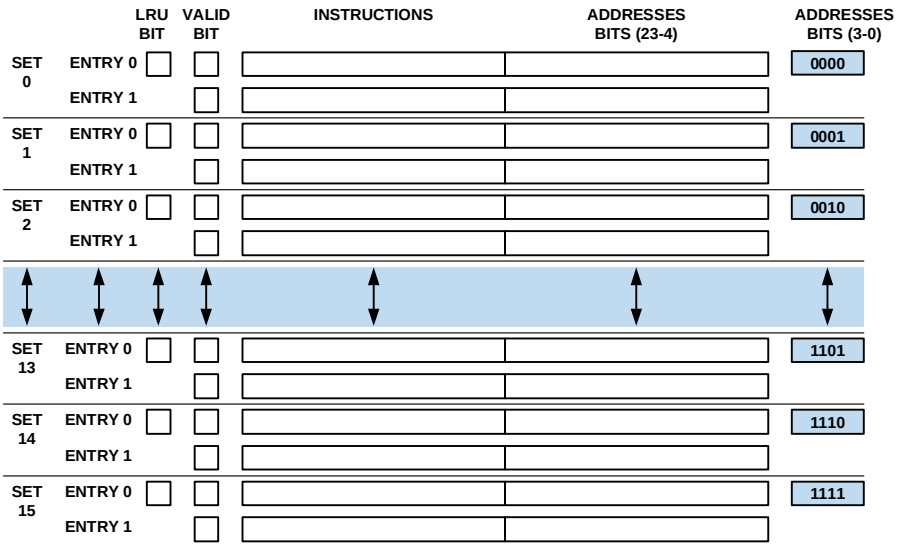


Figure 3-3. Instruction Cache Architecture

Block Conflicts

A bus conflict occurs when an instruction fetch and a data access are made on the same bus. Similarly, a block conflict occurs when multiple accesses are made to the same block in internal memory. This scenario occurs when data is accessed from the same block from which the instructions are

executed. This scenario also occurs when an instruction performs both a DM and PM access to the same block in one instruction. 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.

Using the Cache

After a DSP 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 A-3 on page A-12 lists all 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 by the access. These functions are selected by the CADIS (cache enable/disable) and CAFRZ (cache freeze) bits in the MODE2 register.

If the cache freeze bit of the MODE2 register is set by a program memory data access instruction *n*, then the *n+2* instruction is cached. 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 DSP must wait at least one cycle before executing the PM data access. A program should have a NOP or other non-conflicting instruction inserted after the cache enable instruction.

Optimizing Cache Usage

Cache operation is usually efficient and requires no intervention. However, certain ordering 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 there (that is, a cache hit will occur). However, if another instruction whose address is mapped to the same set displaces this instruction, a cache miss occurs. The LRU bits help to reduce this possibility 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, "hit rate") can go to zero. To solve this problem, 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 3-1. 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 0x103 (into

set 3). Each time the program calls the subroutine, `Inner`, the program memory data accesses at 0x201 and 0x211 displace the instruction at 0x103 by loading the instructions at 0x203 and 0x213 (also into set 3). 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 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 4, instead of set 3.

Table 3-1. Cache Inefficient Code

Address	Instruction
0x0100	lcntr = 1024, do Outer until LCE;
0x0101	r0 = dm(i0,m0), pm(i8,m8) = f3;
0x0102	r1 = r0 – r15;
0x0103	if eq call (Inner);
0x0104	f2 = float r1;
0x0105	f3 = f2 * f2;
0x0106	Outer: f3 = f3 + f4;
0x0107	pm(i8,m8) = f3;
...	
0x0200	Inner: r1 = R13;
0x0201	r14 = pm(i9,m9);
...	
0x0211	pm(i9,m9) = r12;
...	
0x021F	rts;

Branches and Sequencing

One type of nonsequential program flow that the sequencer supports is branching. A branch occurs when a JUMP or CALL/RETURN instruction moves execution to a location other than the next sequential address. For descriptions on how to use JUMP and CALL/RETURN instructions, see the *ADSP-21160 SHARC DSP Instruction Set 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 from an RTS subroutine instruction.
- 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 only pops 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 allows another interrupt to be latched in the IRPTL register and the interrupt mask pointer (IMASKP) register. See Table A-9 on page A-27.
    b. Pops the status stack if the ASTATx/y and MODE1 status registers that 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:

- Branches can be direct or indirect. For direct branches, the sequencer generates the address; for indirect branches, the PM data address generator (DAG2) produces the address
- Direct branches are JUMP or CALL/RETURN 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 a PC-relative address */
```

  Indirect branches are JUMP or CALL/RETURN 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 (i12, m8); /* where (m8,i12) are DAG2 registers */

CALL (i13, m9); /* where (m9,i13) are DAG2 registers */
```

Conditional Branches

The sequencer supports conditional branches. These conditional branches are JUMP or CALL/RETURN 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 3-18. Note that the DSP's Single-Instruction, Multiple-Data (SIMD) mode influences the execution of conditional branches. For more information, see "Summary" on page 3-63.

Delayed Branches

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

As shown in Figure 3-4 and Figure 3-5, the DSP aborts the two instructions after the branch, which are in the fetch and decode stages. For a CALL, the decode address (the address of the instruction after the CALL) is the return address. During the two lost NOP cycles, the pipeline fetches and decodes the first instruction at the branch address.

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

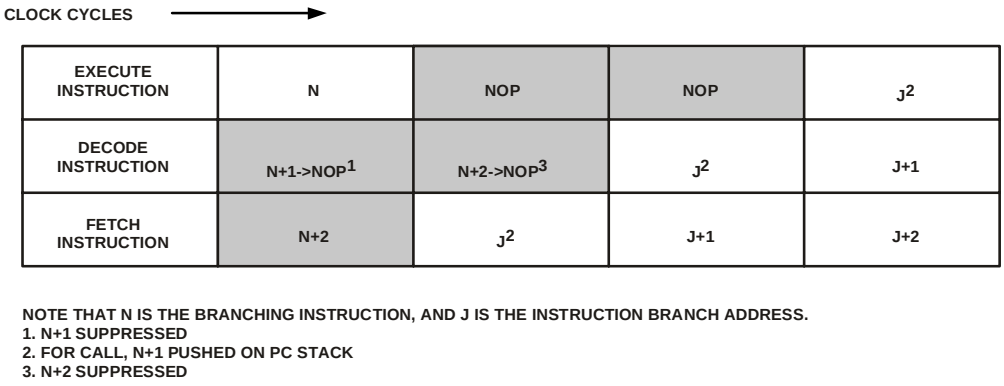


Figure 3-4. Pipelined Execution Cycles for Immediate Branch (Jump/Call)

In delayed branch, JUMP and CALL/RETURN instructions that use the delayed branches (DB) modifier, no instruction cycles are lost in the pipeline. This is because the DSP executes the two instructions after the branch while

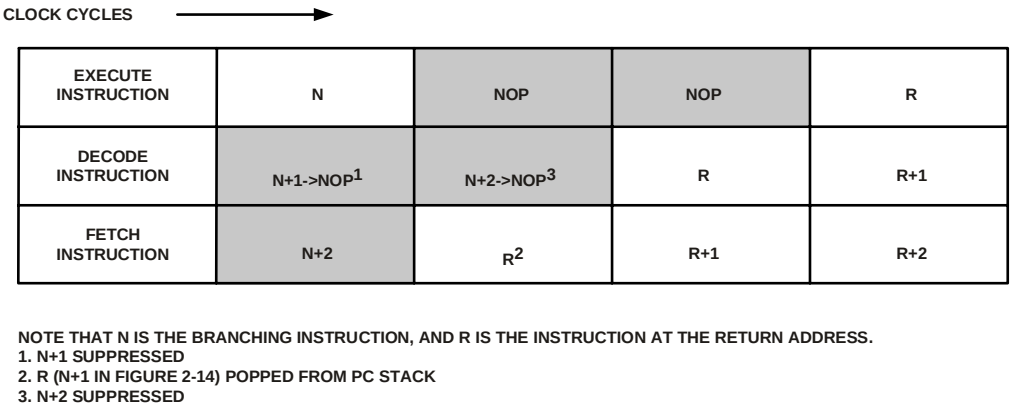


Figure 3-5. Pipelined Execution Cycles for Immediate Branch (return)

the pipeline fills with instructions from the new location. This is shown in the sample code below.

```
call fft1024 (DB);

...

...

jump (pc,10) (DB);
```

As shown in Figure 3-6 and Figure 3-7, the DSP executes the two instructions after the branch, while the instruction at the branch address is fetched and decoded. In the case of a `CALL`, 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 understand because of the instructions between the branch instruction and the actual branch.

CLOCK CYCLES →

EXECUTE INSTRUCTION	N	N+1	N+2	J
DECODE INSTRUCTION	N+1	N+2	J	J+1
FETCH INSTRUCTION	N+2	J[1]	J+1	J+2

NOTE THAT N IS THE BRANCHING INSTRUCTION, AND J IS THE INSTRUCTION BRANCH ADDRESS.
1. FOR A DELAYED BRANCH CALL, N+3 PUSHED ON PC STACK, NOT N+1

Figure 3-6. Pipelined Execution Cycles for Delayed Branch (JUMP or CALL)

CLOCK CYCLES →

EXECUTE INSTRUCTION	N[1]	N+1	N+2	R
DECODE INSTRUCTION	N+1	N+2	R	R+1
FETCH INSTRUCTION	N+2	R	R+1	R+2

NOTE THAT N IS THE BRANCHING INSTRUCTION, AND R IS THE INSTRUCTION AT THE RETURN ADDRESS.
1. R (N+3 PUSHED IN FIGURE 3-5) POPPED FROM PC STACK

Figure 3-7. Pipelined Execution Cycles for Delayed Branch (Return)

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:

- Other branches (no JUMP, CALL, or RETURN instructions)

  Normally, it is not valid to have two conditional instructions that use the (DB) option follow each other. However, where the execution of those instructions is mutually exclusive, it is allowed. For example:

  ```
  if gt jump (PC, 7) (db)
  if le jump (PC,11) (db)
  ```

- Any stack manipulations (no PUSH or POP instructions or writes to the PC stack or PC stack pointer register)
- Any loops or other breaks in sequential operation (no DO/UNTIL or IDLE instructions)

Development software for the DSP should always flag these types of instructions as code errors in the two locations after a delayed branch instruction.

Delayed branches and the instruction pipeline also influence interrupt processing. Because the delayed branch instruction and the two instructions that follow it always execute sequentially, the DSP does not immediately process an interrupt that occurs in 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;
```

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

Loop and Status 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/RETURN instructions) 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 fetch-decode-execute instruction pipeline. It contains the 24-bit address of the instruction the DSP will execute on the next cycle. The PC register, combined with the Program Counter 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 A-5 on page A-19 lists all the bits in the STYKx 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 when 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 a one cycle delay. If the PC stack is overflowed, a write to PCSTKP has no effect. This register can be read from and written to.

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 DSP has transferred all values.

Conditional Sequencing

The sequencer supports conditional execution with conditional logic, as illustrated in Figure 3-19 on page 3-65. 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 3-63.

Each condition that the DSP evaluates has an assembler mnemonic. The condition mnemonics for conditional instructions appear in Table 3-2. 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 (NZ).

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 DSP's system registers, including STKYx and STKYy.

Table 3-2. 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

Table 3-2. IF Condition and DO/UNTIL Termination Mnemonics (Cont'd)

Condition From	Description	True if…	Mnemonic
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
Flag Input	Flag0 asserted	FI0 = 1	FLG0_IN
	Flag0 not asserted	FI0 = 0	NOT FLG0_IN
	Flag1 asserted	FI1 = 1	FLG1_IN
	Flag1 not asserted	FI1 = 0	NOT FLG1_IN
	Flag2 asserted	FI2 = 1	FLG2_IN
	Flag2 not asserted	FI2 = 0	NOT FLG2_IN
	Flag3 asserted	FI3 = 1	FLG3_IN
	Flag3 not asserted	FI3 = 0	NOT FLG3_IN
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:
$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$), or AF and AN, or $\overline{AZ}$ = 0
2 ALU less than (LT) is true if:
$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$), or (AF and AN and $\overline{AZ}$) = 1
3 ALU greater equal (GE) is true if:
$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$), or (AF and AN and $\overline{AZ}$) = 0
4 ALU lesser or equal (LT) is true if:
$\overline{AF}$ and AN xor (AV and $\overline{ALUSAT}$), or $\overline{AF}$ and AN or $\overline{AZ}$ = 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 3-26 and "Restrictions on Short Loops" on page 3-27.

Core Stalls

Like all previous SHARC processors, there are a number of conditions that cause the core to temporarily stop fetching and executing further instructions. This event, known as a *core stall*, occurs when an instruction accesses a peripheral's data-buffer. Specifically, the core stalls when it reads an empty receive buffer or writes a full transmit buffer. Execution resumes once the peripheral moves a valid word of data into the receive buffer or when the peripheral sends one word out from the transmit buffer.

In addition to standard core stall situations, there are four other conditions that cause the ADSP-2126x processor core to stall. The following instructions or sequences of instructions will cause the processor core to

stall for one or more cycles. These stalls were introduced to facilitate the doubling of the core clock rate without modifying the 3-deep instruction-pipeline.

1. Reading or writing any memory mapped register in a conditional instruction stalls the core for one cycle. This means that a total of two cycles are needed for that instruction to complete.

2. Reading the following System/Emulator memory-mapped registers stalls the processor for one cycle. Therefore, a total of two cycles are needed for that instruction.

Register	Address	Register	Address
EEMUIN	0x30020	PSA4E	0x300a7
EEMUSTAT	0x30021	DMA1S	0x300b2
EEMUOUT	0x30022	DMA1E	0x300b3
OSPID	0x30023	DMA2S	0x300b4
SYSCTL	0x30024	DMA2E	0x300b5
BRKCTL	0x30025	D1IC	0x300b6
REVPID	0x30026	D1ID	0x300b7
PSA1S	0x300a0	PMDAS	0x300b8
PSA1E	0x300a1	PMDAE	0x300b9
PSA2S	0x300a2	D2IC	0x300bc
PSA2E	0x300a3	D2ID	0x300bd
PSA3S	0x300a4	EMUN	0x300ae
PSA3E	0x300a5	IOAS	0x300b0
PSA4S	0x300a6	IOAE	0x300b1

3. Reading from all other memory-mapped registers and data-buffers (for example RXSPI, PPCTL, or SPISTAT) stalls the processor core for three cycles. Therefore, a total of four cycles is needed for that instruction to complete.

4. If the following sequence of three instructions are executed without any other instruction between them, the processor stalls for one cycle.

    a. **Instruction 1:** Compute instructions affecting status flags, such as `R2 = R3 - R4;`

    b. **Instruction 2:**Conditional instructions involving post-modify addressing, such as `IF EQ DM(I1,M1) = R15;`

    c. **Instruction 3:** Instructions involving post-modify addressing involving the same I register, such as `R0 = DM(I1,M2);`

    The stall occurs in instruction 2, regardless of whether EQ is true or false. However, the stall only occurs if the 3rd instruction is in the exact sequence and there is no other instruction between 1 and 3.

Loops and Sequencing

Another type of nonsequential program flow that the sequencer supports is looping. A loop occurs when a DO/UNTIL instruction causes the DSP to repeat a sequence of instructions until a condition tests true. Unlike other processors, the SHARC automatically evaluates the loop termination condition and modifies the Program Counter (PC) register appropriately. This allows zero overhead looping.

In addition to the standard status flags available to all conditional instructions (EQ, GT, LT, and so on), a special condition instruction Loop Counter Expired (LCE), is specifically used for terminating loops. This instruction tests whether the loop has completed the required number of iterations in the LCNTR register. Loops that terminate with conditions other than LCE

have some additional restrictions. For more information, see "Restrictions on Ending Loops" on page 3-26 and "Restrictions on Short Loops" on page 3-27. For more information on condition types in DO/UNTIL instructions, see "Interrupts and Sequencing" on page 3-46.

The DSP'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 hardware 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.

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 DSP is executing the instruction that is two locations before the last instruction in the loop (at location $e - 2$, 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 con-

dition 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.

Figure 3-8 and Figure 3-9 show the pipeline states for loop iteration and termination.

CLOCK CYCLES →

EXECUTE INSTRUCTION	E-2[1]	E-1	E	B
DECODE INSTRUCTION	E-1	E	B	B+1
FETCH INSTRUCTION	E	B[2]	B+1	B+2

NOTE THAT E IS THE LOOP END INSTRUCTION, AND B IS THE LOOP START INSTRUCTION.
1. TERMINATION CONDITION TESTS FALSE
2. LOOP START ADDRESS IS TOP OF PC STACK

Figure 3-8. Pipelined Execution Cycles for Loop Back (Iteration)

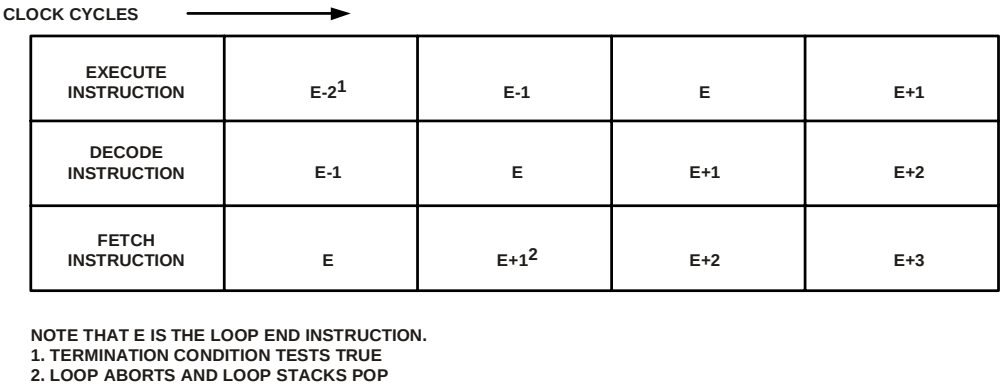


Figure 3-9. Pipelined Execution Cycles for Loop Termination

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.
- 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.
- An instruction that writes to the loop counter from memory cannot be used as the third-to-last instruction of a counter-based loop (at *e–2*, 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/RETURN) 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 and two instruction loops with only one iteration.

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, that is paired with a loop re-entry return, a return (RTS) with loop reentry (LR) modifier. The immediate CALL may be one of the last three instructions of a loop, but not in a one instruction loop or a two instruction, single iteration loop.

Restrictions on Short Loops

The sequencer's pipeline features (which optimize performance in many ways) restrict how short loops iterate and terminate. Short loops (one or two instruction loops) terminate in a special way because they are shorter than the instruction pipeline. Counter-based loops (DO/UNTIL LCE) of one or two instructions are not long enough for the sequencer to check the termination condition two instructions from 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.

Figure 3-10 and Figure 3-11 show the pipeline execution for counter-based single instruction loops. Figure 3-12 and Figure 3-13 show the pipeline execution for counter-based two instruction loops. For no overhead, a loop of length one must be executed at least three times and a loop of length two must be executed at least twice. Loops of length one that iterate only once or twice and loops of length two that iterate only

once incur two cycles of overhead, because two aborted instructions after the last iteration are needed to clear the instruction pipeline.

CLOCK CYCLES →

EXECUTE INSTRUCTION	N[1]	N+1 (PASS 1)	N+1 (PASS 2)	N+1 (PASS 3)	N+2
DECODE INSTRUCTION	N+1	N+1	N+1	N+2	N+3
FETCH INSTRUCTION	N+2	N+1[2]	N+2[3]	N+3	N+4

NOTE: N IS THE LOOP START INSTRUCTION, AND N+2 IS THE INSTRUCTION AFTER THE LOOP.
1. LOOP COUNT (LCNTR) EQUALS 3
2. NO OPCODE LATCH OR FETCH ADDRESS UPDATE; COUNT EXPIRED TESTS TRUE
3. LOOP ITERATION ABORTS; PC AND LOOP STACKS POP

Figure 3-10. Pipelined Execution Cycles for Single Instruction Counter-based Loop with Three Iterations

CLOCK CYCLES →

EXECUTE INSTRUCTION	N[1]	N+1 (PASS 1)	N+1 (PASS 2)	NOP	NOP	N+2
DECODE INSTRUCTION	N+1	N+1	N+1->NOP[4]	N+1->NOP[5]	N+2	N+3
FETCH INSTRUCTION	N+2	N+1[2]	N+1[3]	N+2	N+3	N+4

NOTE: N IS THE LOOP START INSTRUCTION, AND N+3 IS THE INSTRUCTION AFTER THE LOOP.
1. LOOP COUNT (LCNTR) EQUALS 2
2. PC STACK SUPPLIES LOOP START ADDRESS
3. COUNT EXPIRED TESTS TRUE
4. LOOP ITERATION ABORTS; PC AND LOOP STACKS POP

Figure 3-11. Pipelined Execution Cycles for Single Instruction Counter-based Loop with Two Iterations (Two Overhead Cycles)

CLOCK CYCLES →

EXECUTE INSTRUCTION	N[1]	N+1 (PASS 1)	N+2 (PASS 1)	N+1 (PASS 2)	N+2 (PASS 2)	N+3
DECODE INSTRUCTION	N+1	N+2	N+1	N+2	N+3	N+4
FETCH INSTRUCTION	N+2	N+1[2]	N+2[3]	N+3[4]	N+4	N+5

NOTE: N IS THE LOOP START INSTRUCTION, AND N+3 IS THE INSTRUCTION AFTER THE LOOP.
1. LOOP COUNT (LCNTR) EQUALS 2
2. PC STACK SUPPLIES LOOP START ADDRESS
3. COUNT EXPIRED TESTS TRUE
4. LOOP ITERATION ABORTS; PC AND LOOP STACKS POP

Figure 3-12. Pipelined Execution Cycles for Two Instruction Counter-based Loop with Two Iterations

CLOCK CYCLES →

EXECUTE INSTRUCTION	N[1]	N+1 (PASS 1)	N+1 (PASS 1)	NOP	NOP	N+3
DECODE INSTRUCTION	N+1	N+2	N+1->NOP[4]	N+2->NOP[5]	N+3	N+4
FETCH INSTRUCTION	N+2	N+1[2]	N+2[3]	N+3	N+4	N+5

NOTE: N IS THE LOOP START INSTRUCTION, AND N+3 IS THE INSTRUCTION AFTER THE LOOP.
1. LOOP COUNT (LCNTR) EQUALS 1
2. PC STACK SUPPLIES LOOP START ADDRESS
3. COUNT EXPIRED TESTS TRUE
4. LOOP ITERATION ABORTS; PC AND LOOP STACKS POP; N+1 SUPPRESSED
5. N+2 SUPPRESSED

Figure 3-13. Pipelined Execution Cycles for Two Instruction Counter-based Loop with One Iteration (Two Overhead Cycles)

Processing of an interrupt that occurs during the last iteration of a one instruction loop is delayed by one cycle when:

- the loop executes once or twice,
- a two instruction loop executes once, or
- a `NOP` cycle follows one of these loops.

Similarly, in a one instruction loop that iterates at least three times, processing is delayed by one cycle if the interrupt occurs during the third-to-last iteration. For more information on pipeline execution during interrupts, see "Interrupts and Sequencing" on page 3-46.

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

- In a three instruction non counter-based loop, the sequencer tests the termination condition when the DSP executes the top of loop instruction. When the condition tests true, the sequencer completes the iteration of the loop and terminates.
- In a two instruction non counter-based loop, the sequencer tests the termination condition when the DSP executes the last (second) instruction. If the condition becomes true when the first instruction is executed, and the condition tests true during the second instruction, then the sequencer completes one more iteration of the loop before exiting. If the condition becomes true during the second instruction, the sequencer completes two more iterations of the loop before exiting.
- In a one instruction non counter-based loop, the sequencer tests the termination condition every cycle. After the cycle when the condition becomes true, the sequencer completes three more itera-

tions of the loop before exiting. But if the one instruction used in the loop is a PM instruction, then the loop is executed only two more times.

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. LADDR contains the value 0xFFFF FFFF when the loop address stack is empty. "Loop Address Stack Register (LADDR)" on page A-49 lists all the bits in the LADDR register.

The sequencer pushes an entry onto the loop address stack when executing a DO/UNTIL or PUSH loop instruction. The stack entry pops off the stack two 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 A-5 on page A-19. For more information on how these flags work with the loop stacks, see "Loop Counter Stack" on page 3-32. Note that a loop stack overflow causes a maskable interrupt.

Because the sequencer tests the termination condition two 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 at either of these instructions, the value is already the termination address for the next loop stack entry.

Loop Counter Stack

The sequencer's loop support, shown in Figure 3-1 on page 3-3, 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. 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 A-5 on page A-19 lists all the bits in the STYKx register.

Within the sequencer, 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 DSP 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 (CURLCNTR) always contains the current loop count. This register is readable and writable over the DM

data bus. Reading CURLCNTR when the loop counter stack is empty returns the value 0xFFFF FFFF.

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

The loop counter stack pops two instructions before the end of the last loop iteration. When the loop counter stack pops, 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 CURLCNTR does not cause a stack push. If a program writes a new value to CURLCNTR, the program changes the count value of the loop currently executing. 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, some restrictions relating to how a program can write CURLCNTR apply. See "Restrictions on Ending Loops" on page 3-26 for more information.

The next-to-top entry in the loop counter stack (LCNTR) is the location on the stack that takes effect on the next loop stack push. To set up a count value for a nested loop without changing the count for the currently executing loop, a program writes the count value to LCNTR.

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 count stack, making that value the new CURLCNTR value. Figure 3-14 on page 3-34 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 writ-

ten to LCNTR. If a program reads LCNTR during the last two instructions of a terminating loop, the value of LCNTR is the last CURLCNTR value for the loop.

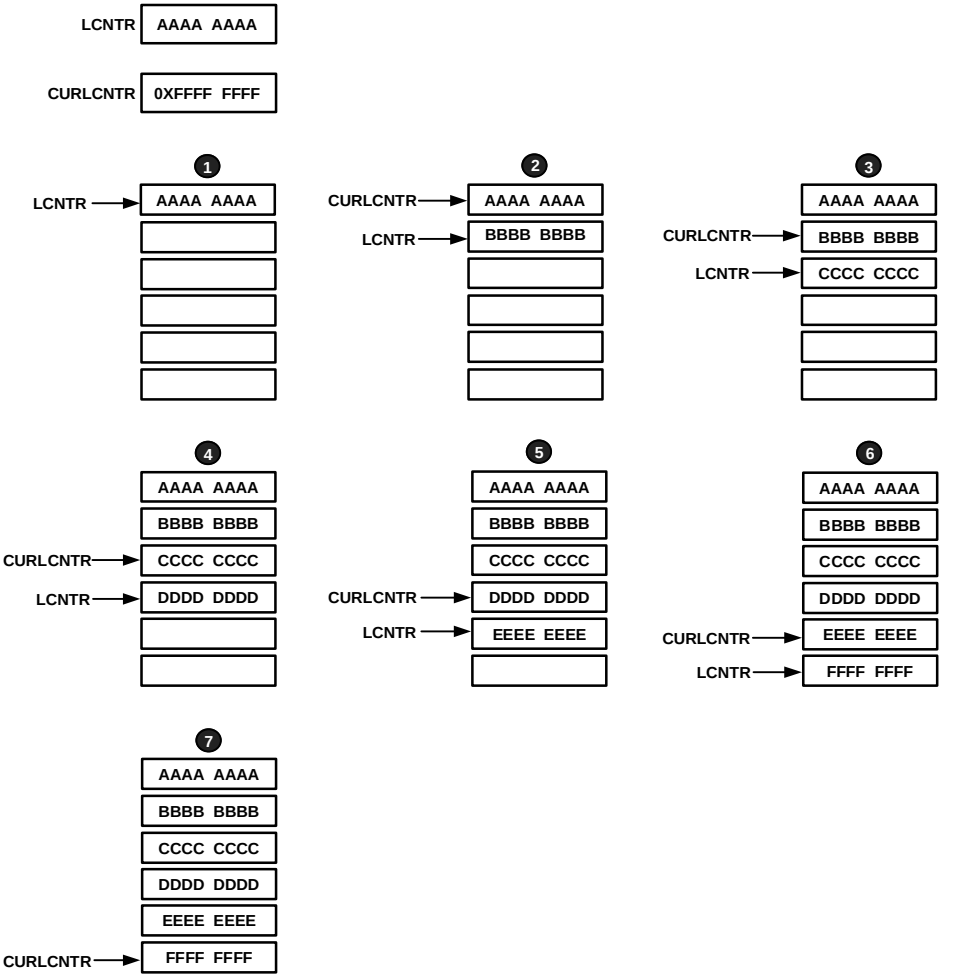


Figure 3-14. Pushing the Loop Counter Stack for Nested Loops

SIMD Mode and Sequencing

The DSP supports a SIMD (Single-Instruction, Multiple-Data) mode. In this mode, both of the DSP'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 DSP records status for the PEx element in the `ASTATx` and `STKYx` registers. The DSP records status for the PEy element in the `ASTATy` and `STKYy` registers. Table A-4 on page A-14 lists the bits in `ASTATx` and `ASTATy`, and Table A-5 on page A-19 lists the bits in `STKYx` and `STKYy`.

Even though the DSP 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 3-3 on page 3-35 summarizes how the sequencer resolves each conditional test when SIMD mode is enabled.

Table 3-3. 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

Table 3-3. Conditional Execution Summary (Cont'd)

Conditional Operation	Conditional Outcome Depends On …
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 uncomplemented 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 uncomplemented register[2])	Executes explicit move to uncomplemented universal register depending on 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 Uncomplemented 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 DSP executes a conditional branch (`JUMP` or `CALL`/`RETURN`) 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 NOPs on the processing element where a branch or loop does not execute. For more information on programming in SIMD mode, see the *ADSP-21160 SHARC DSP Instruction Set 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.

There are four cases for SIMD conditional data moves.

Case #1: Complementary Register Pair Data Move

In this case, data moves from a complementary register pair to a complementary register pair. The DSP 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. For example:

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

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

For this instruction, the DSP 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 3-4.

The moves from the DAG registers to the memory also behave in a similar manner, as demonstrated in Table 3-4. For example:

```
If EQ pm(i0,m0) = m15;
```

Table 3-4. Register-to-Memory 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 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 Move – PEy Explicit Register

For this instruction, the DSP 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 3-5.

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

Example 3: Register-to-Memory Move – PEx Explicit Register

For the following instructions, the DSP is operating in SIMD mode and registers in the PEx data register file are used as the explicit registers. The

Table 3-5. 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 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

data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-6.

```
IF EQ R9 = R2;

IF EQ PX1 = R2;

IF EQ USTAT1 = R2;
```

Table 3-6. 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-Memory Move – PEy Explicit Register

For the following instructions, the DSP 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 3-7.

```
IF EQ R9 = S2;

IF EQ PX1 = S2;

IF EQ USTAT1 = S2;
```

Table 3-7. Register-to-Register Moves – Complementary Register 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
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 uncomplemented register (*Ureg* without a SIMD complement) to a complementary register pair. The DSP executes the explicit move depending on the evaluation of the conditional test in the PEx processing element. The DSP 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.

Example: Register Moves – Uncomplimentary-to-Complementary

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 DSP 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 3-8.

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

Table 3-8. 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 DSP executes the explicit move to the uncomplemented universal register, depending on the condition test in the PEx processing element only. The DSP does not perform an implicit move.

Example: Register Moves – Complementary-to-Uncomplimentary

For all of the following instructions, the DSP 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 3-9.

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

Uncomplemented register to DAG move:

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

DAG to uncomplemented register move:

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

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

DAG to DAG move:

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

Complimented 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 3-9. 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 uncomplemented register with an access to external memory space or IOP memory space. This results in unexpected behavior and should not be used.

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

For the following instructions the DSP is operating in SIMD mode and the explicit register is either a PEx register or PEy register. I0 points to either external memory space or 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. Other than that, if either PEx or PEy is true, transfers to the DAG register occurs. For example:

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

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 indepen-

dent 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.

Timer and Sequencing

The sequencer includes a programmable interval timer, which appears in Figure 3-1 on page 3-3. Bits in the MODE2, TCOUNT, and TPERIOD registers control timer operations as described below.

- **Timer enable** MODE2 Bit 5 (TIMEN). This bit directs the DSP 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 A-3 on page A-12 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 TIMEXP pin. This scenario applies only when TCOUNT is configured as TIMEXP output high for four cycles (when the timer is enabled), as shown in Figure 3-15. 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 clock cycle. If the timer is subsequently disabled, the timer stops decrementing TCOUNT after the next clock cycle as shown in Figure 3-15.

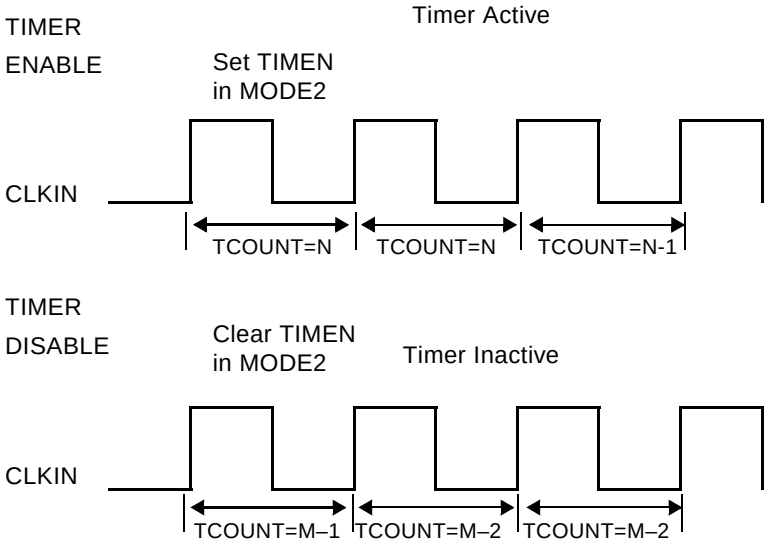


Figure 3-15. Timer Enable and Disable

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 3-57.

As with other interrupts, the sequencer needs two cycles to fetch and decode the first instruction of the timer expired service routine before executing the routine. The pipeline execution for the timer interrupt appears in Figure 3-18 on page 3-50.

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.

Interrupts and Sequencing

Another type of nonsequential program flow that the sequencer supports is interrupt processing. 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 DSP 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 Vector Addresses" on page B-1.

The DSP supports three prioritized, individually-maskable external interrupts, each of which can be either level- or edge-sensitive. External interrupts occur when another device asserts one of the DSP's interrupt inputs ($\overline{\text{IRQ}}$2-0). The DSP also supports internal interrupts. An internal interrupt can stem from arithmetic exceptions, stack overflows, DMA completion and/or peripheral data buffer status, or circular data buffer

overflows. Several factors control the DSP's response to an interrupt. The DSP responds to an interrupt request if:

- the DSP 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 DSP responds to an interrupt, the sequencer branches program execution with a call to the corresponding interrupt vector address. Within the DSP'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. Each interrupt vector has associated latch and mask bits. For a list of interrupt vector addresses and their associated latch and mask bits, see Table B-2 on page B-2.

"Interrupt Latch Register (IRPTL)" on page A-25, "Interrupt Register (LIRPTL)" on page A-42, and "Interrupt Mask Register (IMASK)" on page A-30 lists the latch and mask bits.

To process an interrupt, the DSP'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.

At the end of the interrupt service routine (ISR), the sequencer processes the return-from-interrupt (RTI) instruction and performs the steps shown below. Between servicing and returning, the sequencer clears the latch bit of the in-progress ISR every cycle until the RTI is executed. This prevents the same interrupt from recurring until the ISR is done. Refer to the JUMP (CI) code example on page 3-62 to learn how to prevent this clearing.

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).

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 two cycles of branching to the interrupt vector follow the recognition cycle.

The DSP responds to interrupts in three stages: synchronization and latching (1 cycle), recognition (1 cycle), and branching to the interrupt vector (2 cycles). Figure 3-16, Figure 3-18, and Figure 3-17 show the pipelined execution cycles for interrupt processing.

For most interrupts, both internal and external, only one instruction is executed after the interrupt occurs (and before the two aborted instructions), while 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 three cycle latency associated with the $\overline{\text{IRQ2-0}}$ interrupts. If an interrupt is latched by

CLOCK CYCLES →

EXECUTE INSTRUCTION	N-1[1]	N	NOP	NOP	V
DECODE INSTRUCTION	N	N+1->NOP[3]	N+2->NOP[5]	V	V+1
FETCH INSTRUCTION	N+1	N+2[2]	V[4]	V+1	V+2

NOTE THAT N IS THE SINGLE CYCLE INSTRUCTION, AND V IS THE INTERRUPT VECTOR INSTRUCTION.
1. INTERRUPT OCCURS
2. INTERRUPT RECOGNIZED
3. N+1 PUSHED ON PC STACK; N+1 SUPPRESSED
4. INTERRUPT VECTOR OUTPUT
5. N+2 SUPPRESSED

Figure 3-16. Pipelined Execution Cycles for Interrupt During Single Cycle Instruction

CLOCK CYCLES →

EXECUTE INSTRUCTION	N-1[1]	N	N+1	N+2	NOP	NOP	V
DECODE INSTRUCTION	N	N+1	N+2	J->NOP[4]	J+1->NOP[6]	V	V+1
FETCH INSTRUCTION	N+1	N+2[2]	J	J+1[3]	V[5]	V+1	V+2

NOTE THAT N IS THE DELAYED BRANCH INSTRUCTION, J IS THE INSTRUCTION AT THE BRANCH ADDRESS, AND V IS THE INTERRUPT VECTOR INSTRUCTION.
1. INTERRUPT OCCURS
2. INTERRUPT RECOGNIZED, BUT NOT PROCESSED
3. INTERRUPT PROCESSED
4. FOR A CALL, N+3 (RETURN ADDRESS) IS PUSHED ONTO THE PC STACK; J SUPPRESSED
5. INTERRUPT VECTOR OUTPUT
6. J PUSHED ON PC STACK; J+1 SUPPRESSED

Figure 3-17. Pipelined Execution Cycles for Interrupt During Delayed Branch Instruction

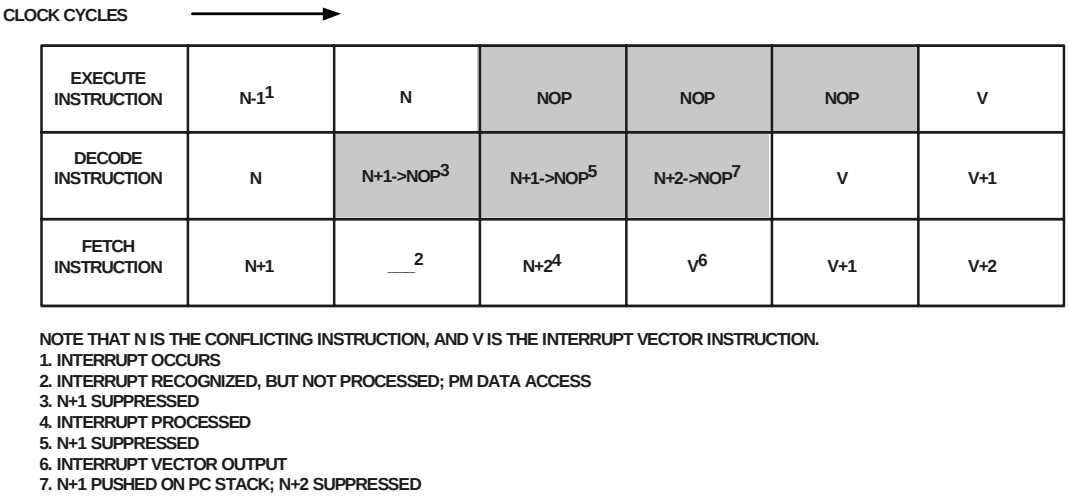


Figure 3-18. Pipelined Execution Cycles for Interrupt During Instruction with Conflicting PM Data Access (Instruction not Cached)

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 3-60.

Certain DSP operations that span more than one cycle hold off interrupt processing. If an interrupt occurs during one of these operations, the DSP

latches the interrupt, but delays its processing. The operations that have delayed interrupt processing are:

- A branch (JUMP or CALL/RETURN) instruction and the following cycle, whether it is an instruction (in a delayed branch) or a NOP (in a non-delayed branch)
- 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
- The third-to-last iteration of a one instruction loop
- The last iteration of either a one instruction loop executed once or twice or a two instruction loop executed once, and the following cycle (which is a NOP)
- The first of the two cycles used to fetch and decode the first instruction of an interrupt service routine
- Any wait states for external memory accesses

Sensing Interrupts

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

The DSP 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 DSP samples it after returning from its service routine, the DSP treats the signal as a new request. The DSP repeats the same interrupt routine without returning to the main program, assuming no higher priority interrupts are active.

The DSP 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 ($\overline{\text{IRQ0E}}$) directs the DSP to detect $\overline{\text{IRQ0}}$ as edge-sensitive (if 1) or level-sensitive (if 0).
- **Interrupt 1 Sensitivity.** Bit 1 ($\overline{\text{IRQ1E}}$) directs the DSP to detect $\overline{\text{IRQ1}}$ as edge-sensitive (if 1) or level-sensitive (if 0).
- **Interrupt 2 Sensitivity.** Bit 2 ($\overline{\text{IRQ2E}}$) directs the DSP to detect $\overline{\text{IRQ2}}$ as edge-sensitive (if 1) or level-sensitive (if 0).

Table A-3 on page A-12 lists all of the bits in the MODE2 register.

The DSP accepts external interrupts that are asynchronous to the DSP'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 DSP 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-2126x data sheet.

Masking Interrupts

The sequencer supports interrupt masking—latching an interrupt, but not responding to it. Except for the RESET and EMU interrupts, all interrupts are maskable. If a masked interrupt is latched, the DSP 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 3-10.

Table 3-10. ADSP-2126x Interrupts

Latch Register [Bit]	Mask Register [Bit]	Mask Pointer Register [Bit]	Vector Address	Interrupt Name	Function
IRPTL [0] EMUIL	N/A	N/A	0x00	EMUI	Emulator (read-only, non-maskable) HIGHEST PRIORITY
IRPTL [1] RSTIL	N/A	N/A	0x04	RSTI	Reset (read-only, non-maskable)
IRPTL [2] IICDIL	IMASK [2] IICDIMSK	IMASKP [2] IICDIMSKP	0x08	IICDI	Illegal Input Condition Detected
IRPTL[3] SOVFIL	IMASK [3] SOVFIMSK	IMASKP [3] SOVFIMSKP	0x0C	SOVFI	Status loop or mode stack overflow; or PC stack full
IRPTL [4] TMZHIL	IMASK [4] TMZHIMSK	IMASKP [4] TMZHIMSKP	0x10	TMZHI	Timer=0 (high priority option)
IRPTL [5]	IMASK [5]	IMASKP [5]	0x14		Reserved
IRPTL [6] BKPIL	IMASK [6] BKPIMSK	IMASKP [6] BKPIMSKP	0x18	BKPI	Hardware Breakpoint Interrupt
IRPTL [7]	IMASK [7]	IMASKP [7]	0x1C		Reserved

Table 3-10. ADSP-2126x Interrupts (Cont'd)

Latch Register [Bit]	Mask Register [Bit]	Mask Pointer Register [Bit]	Vector Address	Interrupt Name	Function
IRPTL [8] IRQ2IL	IMASK [8] IRQ2IMSK	IMASKP [8] IRQ2IMSKP	0x20	$\overline{\text{IRQ2}}$I	Interrupt associated with $\overline{\text{IRQ2}}$ pin
IRPTL [9] IRQ1IL	IMASK [9] IRQ1IMSK	IMASKP [9] IRQ1IMSKP	0x24	$\overline{\text{IRQ1}}$I	Interrupt associated with $\overline{\text{IRQ1}}$ pin
IRPTL [10] IRQ0IL	IMASK [10] IRQ0IMSK	IMASKP [10] IRQ0IMSKP	0x28	$\overline{\text{IRQ0}}$I	Interrupt associated with $\overline{\text{IRQ0}}$ pin
IRPTL [11] DAIHIL	IMASK [11] DAIHIMSK	IMASKP [11] DAIHIMSKP	0x2C	DAIHI	DAI high priority interrupt
IRPTL [12] SPIHIL	IMASK [12] SPIHIMSK	IMASKP [12] SPIHIMSKP	0x30	SPIHI	SPI Transmit or Receive (higher priority option)
IRPTL [13] GPTMR0IL	IMASK [13] GPTMR0IMSK	IMASKP [13] GPTMR0IMSKP	0x34	GPTMR0I	General-purpose IOP Timer 0 Interrupt
IRPTL [14] SP1IL	IMASK [14] SP1IMSK	IMASKP [14] SP1IMSKP	0x38	SP1I	SPORT1 Interrupt
IRPTL [15] SP3IL	IMASK [15] SP3IMSK	IMASKP [15] SP3IMSKP	0x3C	SP3I	SPORT3 Interrupt
IRPTL [16] SP5IL	IMASK [16] SP5IMSK	IMASKP [16] SP5IMSKP	0x40	SP5I	SPORT5 Interrupt
LIRPTL [0] SP0IL	LIRPTL [10] SP0IMSK	LIRPTL [20] SP0IMSKP	0x44	SP0I	SPORT0 Interrupt
LIRPTL [1] SP2IL	LIRPTL [11] SP2IMSK	LIRPTL [21] SP2IMSKP	0x48	SP2I	SPORT2 Interrupt
LIRPTL [2] SP4IL	LIRPTL [12] SP4IMSK	LIRPTL [22] SP4IMSKP	0x4C	SP4I	SPORT4 Interrupt

Table 3-10. ADSP-2126x Interrupts (Cont'd)

Latch Register [Bit]	Mask Register [Bit]	Mask Pointer Register [Bit]	Vector Address	Interrupt Name	Function
LIRPTL [3] PPIL	LIRPTL [13] PPIMSK	LIRPTL [23] PPIMSKP	0x50	PPI	Parallel Port Interrupt
LIRPTL [4] GPTMR1IL	LIRPTL [14] GPTMR1IMSK	LIRPTL [24] GPTMR1IMSKP	0x54	GPTMR1I	General-purpose IOP Timer 1 Interrupt
LIRPTL [5]	LIRPTL [15]	LIRPTL [25]	0x58	--	Reserved
LIRPTL [6] DAILIL	LIRPTL [16] DAILIMSK	LIRPTL [26] DAILIMSKP	0x5C	DAILI	DAI Low Priority Interrupt
LIRPTL [7] IRPTL [17, 18, 19]	LIRPTL [17] IMASK [19:17]	LIRPTL [27] IMASKP [19:17]	0x60 0x64-0x6F	-- --	Reserved Reserved
LIRPTL [8] GPTMR2IL	LIRPTL [18] GPTMR2IMSK	LIRPTL [28] GPTMR2IMSKP	0x70	GPTMR2I	General-purpose IOP Timer 2 Interrupt
LIRPTL [9] SPILIL	LIRPTL [19] SPILIMSK	LIRPTL [29] SPILIMSKP	0X74	SPILI	SPI Transmit or Receive (lower priority option)
IRPTL [20] CB7IL	IMASK [20] CB7IMSK	IMASKP [20] CB7IMSKP	0x78	CB7I	Circular Buffer 7 Overflow
IRPTL [21] CB15IL	IMASK [21] CB15IMSK	IMASKP [21] CB15IMSKP	0x7C	CB15I	Circular Buffer 15 Overflow
IRPTL [22] TMZLIL	IMASK [4] TMZLIMSK	IMASKP [22] TMZLIMSKP	0x80	TMZLI	Timer=0 (Low Priority Option)
IRPTL [23] FIXIL	IMASK [23] FIXILMSK	IMASKP [23] FIXIMSKP	0x84	FIXI	Fixed-point Overflow
IRPTL [24] FLTOIL	IMASK [24] FLTOIMSK	IMASKP [24] FLTOIMSKP	0x88	FLTOI	Floating-point Overflow Exception

Table 3-10. ADSP-2126x Interrupts (Cont'd)

Latch Register [Bit]	Mask Register [Bit]	Mask Pointer Register [Bit]	Vector Address	Interrupt Name	Function
IRPTL [25] FLTUIL	IMASK [25] FLTUIMSK	IMASKP [25] FLTUIMSKP	0x8C	FLTUI	Floating-point Underflow Exception
IRPTL [26] FLTIIL	IMASK [25] FLTIIMSK	IMASKP [26] FLTIIMSKP	0x90	FLTII	Floating-point Invalid Exception
IRPTL [27] EMULIL	IMASK [27] EMULIMSK	IMASKP [27] EMULIMSKP	0x94	EMULI	Emulator low priority interrupt
IRPTL [28] SFT0IL	IMASK [28] SFT0IMSK	IMASKP [28] SFT0IMSKP	0x98	SFT0I	User software interrupt 0
IRPTL [29] SFT1IL	IMASK [29] SFT1IMSK	IMASKP [29] SFT1IMSKP	0x9C	SFT1I	User software interrupt 1
IRPTL [30] SFT2IL	IMASK [30] SFT2IMSK	IMASKP [30] SFT2IMSKP	0xA0	SFT2I	User software interrupt 2
IRPTL [31] SFT3IL	IMASK [31] SFT3ISMK	IMASKP [31] SFT3ISMKP	0xA4	SFT3I	User software interrupt 3: LOWEST PRIORITY

Table A-2 on page A-5 lists all of the bits in MODE1, Table A-10 on page A-32 lists all of the bits in IMASK, and Table A-12 on page A-44 lists all of the bits in LIRPTL.

All interrupts are masked at reset except for the non-maskable and boot interrupts. For booting, the DSP automatically unmasks and uses the parallel port interrupt (PPI) or high priority SPI port (SPIHI) interrupt after reset. Usage depends on whether the ADSP-2126x processor is booting from EPROM, or an SPI master or slave. See also the product specific peripherals manual for a description of DAI interrupts.

Latching Interrupts

When the DSP recognizes an interrupt, the DSP's interrupt latch (IRPTL and LIRPTL) registers set a bit (latch) to record that the interrupt occurred. The bits in these registers indicate all interrupts that are currently being serviced or are pending. 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 once that interrupt is serviced. Throughout the execution of the interrupt's service routine, the DSP 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 3-62.

The interrupt latch bits in IRPTL correspond to interrupt mask bits in the IMASK register. In both registers, the interrupt bits are arranged in order of priority. The interrupt priority is from 0 (highest) to 31 (lowest). Interrupt priority determines which interrupt is serviced first when more than one occurs in the same cycle. Priority also determines which interrupts are nested when the DSP has interrupt nesting enabled. For more information, see "Nesting Interrupts" on page 3-60.

While the IRPTL register latches interrupts for a variety of events, the LIRPTL register contains latch and mask bits for the SP0, SP2, SP4, PP, GPTMR1, GPTMR2, DAI (low priority), SPI (low priority) interrupts.

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 A-5 on page A-19 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 DSP services the higher priority interrupt first, then it services the lower priority interrupt.

The IRPTL register also supports 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 DSP 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 DSP 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—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 A-5 on page A-19 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 DSP 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 A-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 3-62. 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 the 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 DSP to enable (if 1) or disable (if 0) interrupt nesting.
- **Interrupt Mask Pointer.** IMASKP bits. These bits list the interrupts in priority order and provide a temporary interrupt mask for each nesting level.
- **SPI Port DMA Transmit or Receive Interrupt Mask Pointer.** LIRPTL Bit 29 (SPILIMSKP). This bit is for the SPI port transmit or receive DMA interrupt. It provides a temporary interrupt mask.
- **General-Purpose IOP Timer Interrupt Mask Pointer.** LIRPTL Bits 24 and 28 (GPTMR1MSKP and GPTMR2MSKP). These bits are for the general purpose IOP timer 1 and timer 2 interrupts, respectively. They provide a temporary interrupt mask.
- **Serial Port Interrupt Mask Pointer.** LIRPTL Bits 22-20 (SPxMSKP). These bits are for the serial port interrupts (SP0, SP2, and SP4). They provide a temporary interrupt mask.
- **DAI Low Priority Interrupt Mask Pointer.** LIRPTL Bit 26 (DAILIMSKP). This bit is for the DAI low priority interrupt. It provides a temporary interrupt mask.

Table A-2 on page A-5 lists all of the bits in MODE1, Table A-11 on page A-38 lists all of the bits in IMASKP, and Table A-12 on page A-44 lists all of the bits in LIRPTL.

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 DSP processes them according to their priority after the nested routines finish.

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 DSP uses the interrupt mask pointer (IMASKP) to create a temporary interrupt mask for each level of interrupt nesting; the IMASK value is not effected. The DSP changes IMASKP each time a higher priority interrupt interrupts a lower priority service routine.

The bits in IMASKP correspond to the interrupts in order of priority. When an interrupt occurs, the DSP sets its bit in IMASKP. If nesting is enabled, the DSP 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 DSP clears the highest priority bit set in IMASKP and generates a new temporary interrupt mask.

The DSP 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 IMASKP register are for interrupt controller use only. Modifying these bits interferes with the proper operation of the interrupt controller. Furthermore, explicit bit manipulation of *any* bit in the LIRPTL register while the IRPTEN bit (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—the DSP clears the bit during every cycle, preventing 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 DSP 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 3-26. 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,3) (DB,CI); /*Clear interrupt status*/

instr3;

instr4;

instr5;

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

The JUMP (PC,3)(DB,CI) instruction only continues linear execution flow by jumping to the location PC + 3 (instr5). The two intervening instructions (instr3, instr4) are executed 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 DSP in IDLE—a special instruction that halts the processor core in a low power state. The halt occurs until any interrupt is latched, serviced, and then returned from using the RTI instruction. When executing an IDLE instruction, the sequencer fetches one more instruction at the current fetch address and then suspends operation. The DSP'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 interrupt occurs, the processor responds normally. After two cycles used to fetch and decode the first instruction of the interrupt service routine, the processor continues executing 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 DSP 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 3-19 identifies all the functional blocks and their relationship to one another in detail.

Table 3-11 and Table 3-12 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 address, decode address, 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), see the *ADSP-21160 SHARC DSP Instruction Set 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.

With the lists of sequencer and system registers, Table 3-11 and Table 3-12 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" 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.

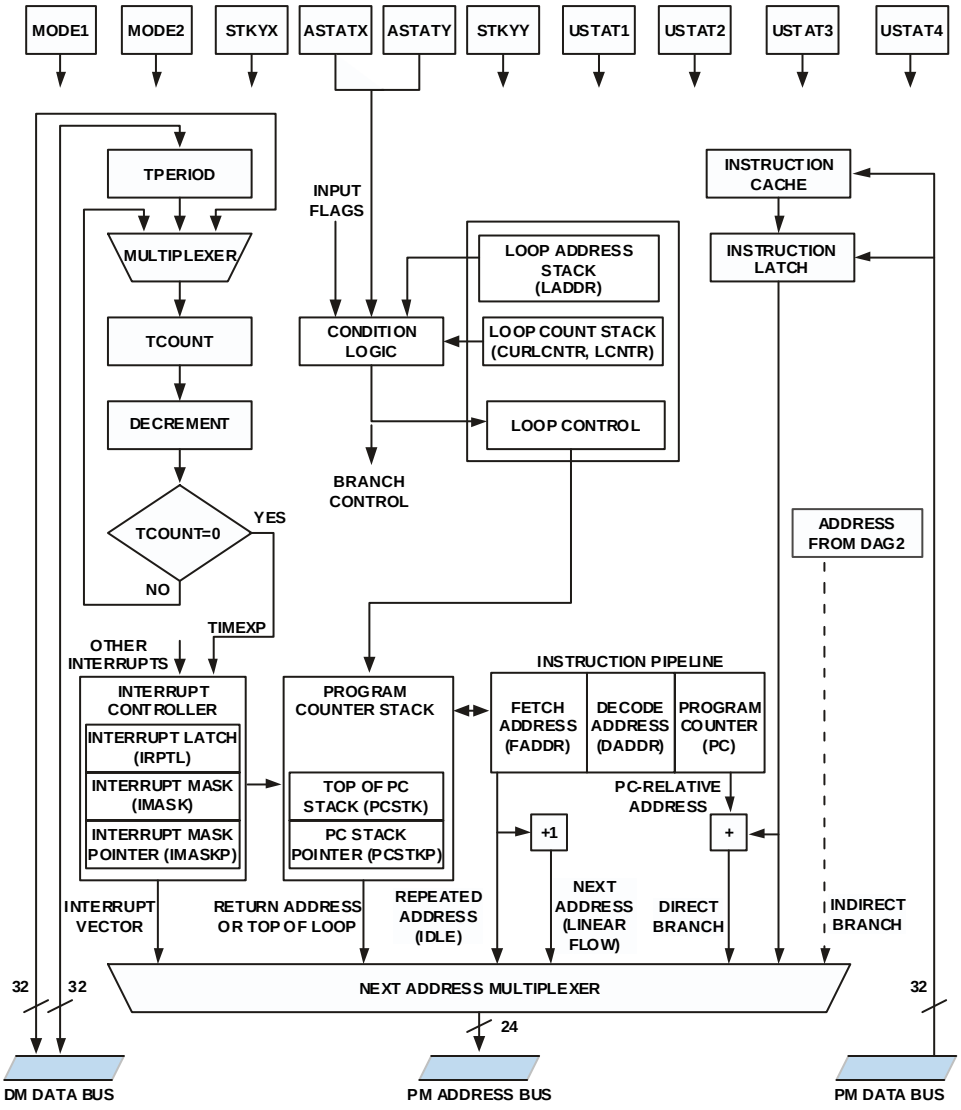


Figure 3-19. Program Sequencer Block Diagram

Table 3-11. 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 3-12. 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

Table 3-12. System Registers Read and Effect Latencies (Cont'd)

Register	Contents	Bits	Read Latency	Effect Latency
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

Summary

4 DATA ADDRESS GENERATORS

The DSP'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.

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 4-14.

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).

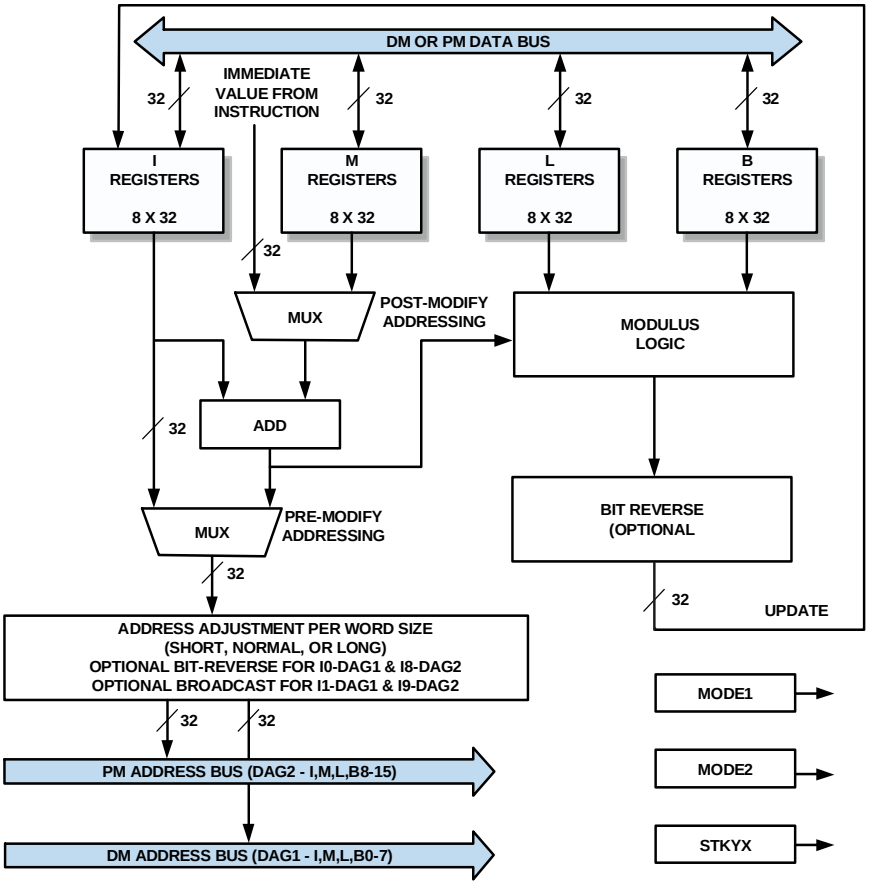


Figure 4-1. Data Address Generator (DAG) Block Diagram

- **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 A-2 on page A-5 lists all of the bits in MODE1.

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 MODE1 is cleared (= 0) at reset. This makes the ADSP-2126x processor code compatible with the ADSP-2106x SHARC family (ADSP-21060/1/2 and ADSP-21065L) where circular buffering is active upon reset.

Note also that circular buffering is disabled upon reset for the ADSP-21160 processor when porting code from an ADSP-21160 processor to an ADSP-2126x processor.

For more information on setting up and using circular buffers, see "Addressing Circular Buffers" on page 4-14. When using circular buffers, the DAGs can generate an interrupt on buffer overflow (wraparound). For more information, see "Using DAG Status" on page 4-8.

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. Note also 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.

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}

Table 4-1. Dual Processing Element Register Load Broadcasts (Cont'd)

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}

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 DSP 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 DSP operations. Note that there is a maximum 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 DAGs' primary and alternate register sets.

To share data between contexts, a program places the data to be shared in one half of either the current DAGs' registers or the other DAG's registers and activates the alternate register set of the other half. The following

example demonstrates how code handles the maximum one cycle of latency from the instruction that sets the bit in MODE1 to when the alternate registers may be accessed. Note that programs should use a NOP instruction for the wait period.

```
BIT SET MODE1 SRD1L; /* Activate alternate dag1 lo regs */

NOP;                 /* Wait for access to alternates */

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 only bit-reverse 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 */

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 DSP 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 the *ADSP-21160 SHARC DSP Instruction Set Reference*.

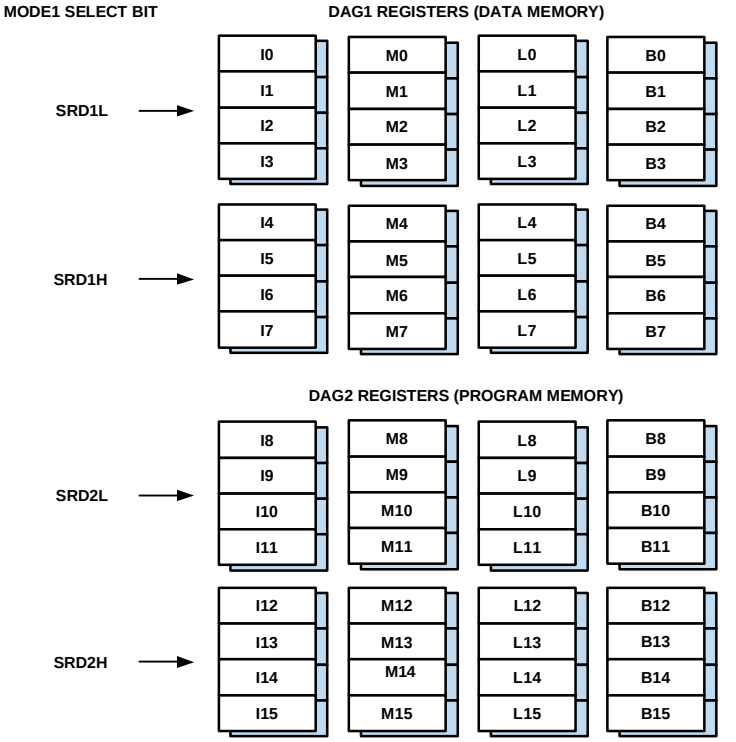


Figure 4-2. Data Address Generator Primary and Alternate Registers

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 circles past the buffer's base address. (See "Addressing Circular Buffers" on page 4-14.)

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), 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 DSP's DAGs perform several types of operations to generate data addresses. As shown in Figure 4-1, the DAG registers and the MODE1, MODE2, and STKYx registers all contribute to DAG operations. The following sections provide details on DAG operations:

- "Addressing With DAGs" on page 4-10
- "DAG Pre-Modify Addressing" on page 4-12
- "Pre-Modify Locking" on page 4-13

- “Addressing Circular Buffers” on page 4-14
- “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/external) all influence data access operations.

Addressing With DAGs

The DAGs support two types of modified addressing which is defined as generating 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. The other type of modified addressing is post-modify addressing. 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 4-2 on page 4-8 compares pre- and post-modify addressing.

The difference between pre-modify and post-modify instructions in the DSP’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 */
```

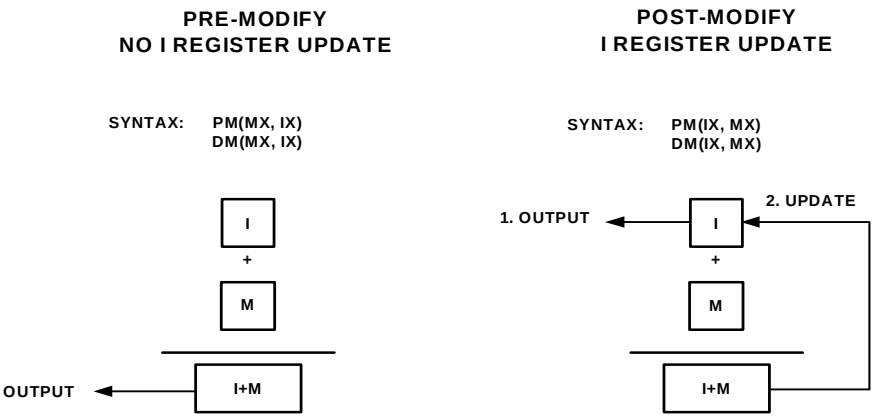


Figure 4-3. Pre-Modify and Post-Modify Operations

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 4-8.

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 */
```

Note that pre-modify addressing operations must not change the memory space of the address.

DAG Pre-Modify Addressing

The pre-modify addressing scheme in the ADSP-2126x processor uses an `I` register cache to speed up address computation. One 32-bit register is provided with each DAG. When a DAG pre-modify access is performed for the first time, the `I` register value is recorded in the `I` register cache and replayed for all subsequent pre-modify accesses using the same `I` register. The first access takes two cycles, and the subsequent accesses are completed in the same cycle. If another pre-modify instruction using a different `I` register is executed, the new `I` register is recorded and the previous value in the cache is replaced. Again, this access takes two cycles.

For example,

```
DM(0x1,Ia) <-> Reg;  /* Takes two cycles */

...                  /* Non pre-modify instructions */

DM(0x4,Ia) <-> Reg;  /* Takes one cycle */

...

DM(0x4,Ib) <-> Reg;  /* Ib loaded … Ia replaced … Two cycles */
```

The pre-modify cache register loading is not supported by legacy products.

The `M` register path does not contain a cache. Therefore, if an instruction of the following type,

```
DM(Ma,Ib) <-> Reg;  /* Would always take two cycles */
```

is executed, then the instruction takes two cycles. The Ib instruction is recorded into the I register cache.

The DSP invalidates the I register cache and replaces the data if any write to that I register is performed. Writes can be in the form of 32-bit or 64-bit writes or updates in the form of post-modify instructions.

When accessing memory indirectly and using pre-modify addressing, do not cross memory bank boundaries.

Pre-Modify Locking

In order to avoid frequent replacement of the cached I register, any register presently in the I register cache can be locked (in other words, the I register content is not replaced in favor of any other pre-modify instruction) using the DAG1 and DAG2 lock bits in the MODE1 register. If an update or a write happens to the locked I register, then the cache is invalid. However, the cache will still remember the locked I register and will subsequently load only that I register when a pre-modify occurs with that I register. All pre-modify instructions with other I registers consume two processor cycles.

Type 9 and Type 10 instructions in DAG2 do not use the ICACHE for indirect branches. The previous ICACHE content (if any) is not replaced, and the instructions take two processor cycles. See the *ADSP-21160 SHARC DSP Instruction Set Reference* for more information on these instruction types.

Data Addressing Stalls

As explained in the previous sections, the instruction sequence stalls for one cycle if a read-after-write hazard is detected on a DAG register. For example, the following sequence automatically generates a one cycle stall.

```
I0 = R0;

DM(I0,M0) <-> R1;
```

Pre-modify addressing causes stalls if there is a miss on the I register cache. Pre-modify stalls are explained in detail in the section on "DAG Pre-Modify Addressing" on page 4-12.

DAG conditional addressing can generate stalls if a post-modify instruction is aborted.

```
R2 = R3 - R4;             /* Compute setting flags */

IF EQ DM(I1,M1) <-> R1;   /* Flag is used immediately */

DM(I1,M2) <-> R2;         /* Updated I1 is used immediately */
```

If the second instruction finds its condition true, then no stalls are inserted. However, if the second instruction is annulled because the condition was false, then a stall is inserted in the address computation (decode) stage of the third instruction. Note that a stall is generated only if the above sequence is executed back-to-back.

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 from or adds to the length of the buffer value, wrapping the index pointer back to the start 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.

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 DSP's memory map, see Figure 4-1 on page 4-3.

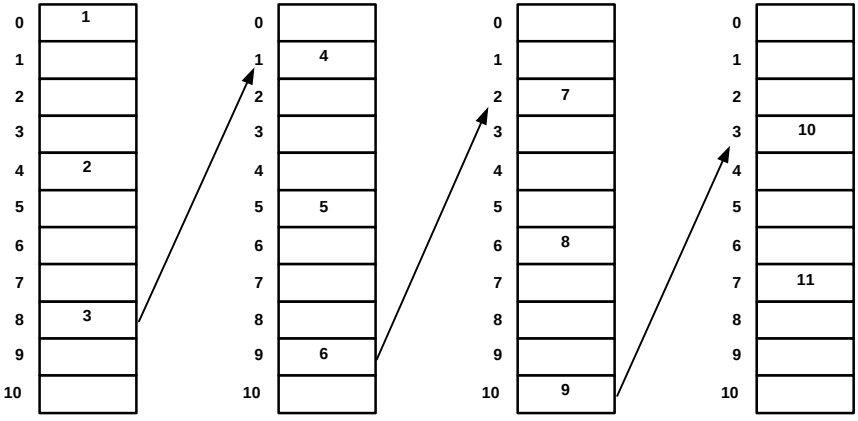
As shown in Figure 4-4, programs use the following steps to set up a circular buffer:

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; SET BY DEFAULT */
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 */
```



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

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.

On the ADSP-2126x 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.

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 the buffer length, the DAG writes the value to the I register. If the updated value is outside the buffer length, the DAG subtracts (positive) or adds (negative) 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 (end of buffer)

- If M is negative:

$$I_{new} = I_{old} + M \text{ if } I_{old} + M \geq \text{buffer base (start of buffer)}$$

$$I_{new} = I_{old} + M + L \text{ if } I_{old} + M < \text{buffer base (start of buffer)}$$

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 amount (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 4-8.

When a program needs to use I7 or I15 without circular buffering and the DSP 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 DSP 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 a long word access, SIMD access, or normal word access (with LW option) crosses the end of the circular buffer, the DSP 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 DSP'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 DSP's 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.

Programs should use care in the case where the DAG register is a destination of a transfer from a register file data register source. Programs should use a conditional operation to select either one processing element or nei-

ther 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.

In the case where a DAG register is both source and destination, the data move operation executes the same as it would if SIMD mode were disabled (PEYEN cleared).

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 4-5 illustrates these transfers.

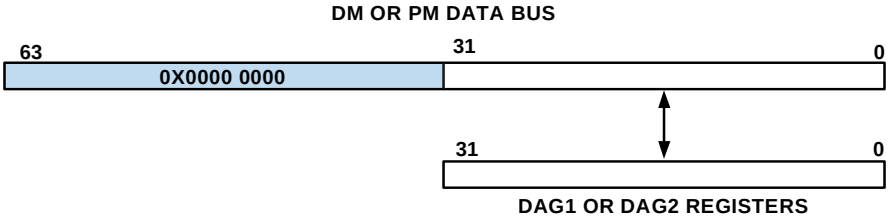


Figure 4-5. Normal Word (32-bit) DAG Register Memory Transfers

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 4-6 illustrates these transfers.

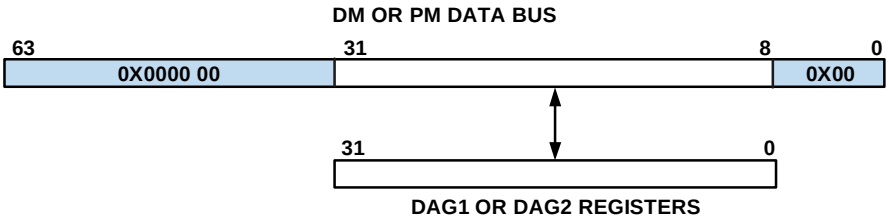


Figure 4-6. DAG Register-to-Data Register Transfers

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 4-7 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, as for example:

`I0 = PX;` equates to `I0 = PX1;`

only the contents of the PX1 register are written into I0. However, the following example:

`PX = I0;` equates to `PX1 = PX2 = I0;`.

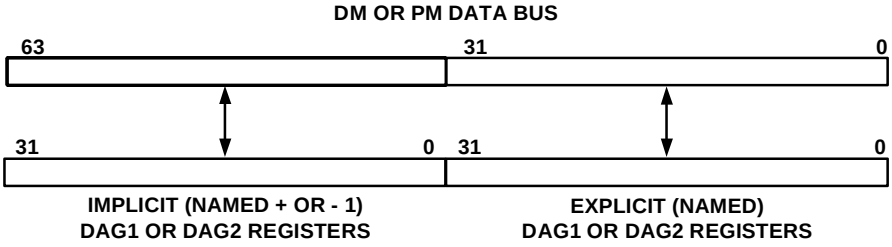


Figure 4-7. Long Word DAG Register-to-Data Register Transfers

DAG Register Transfer Restrictions

The two types of transfer restrictions are hold-off conditions and illegal conditions that the DSP does not detect.

For certain instruction sequences involving transfers to and from DAG registers, an extra (`NOP`) cycle is automatically inserted by the processor. In case where an instruction that loads a DAG register is followed by an instruction that uses any register in the same DAG register pair[1] for data addressing, modify instructions, or indirect jumps, the DSP inserts an extra (`NOP`) cycle between the two instructions. This hold-off occurs because the same bus is needed by both operations in the same cycle. Therefore, the second operation must be delayed. The following example causes a delay because it exhibits a write/read dependency in which `I0` is written in one cycle. The results of that register write are not available to a register read for one cycle. Note that if either instruction had specified `I1`, the stall occurs only if the first instruction performs a long word (LW)

[1] DAG registers are accessible in pair granularity for single cycle access. The pairings are odd-even. For example I0 and I1 are a pair, and I2 and I3 are a pair.

access. The DAG detects write/read dependencies with a register pair granularity:

```
I0 = 8;
DM(I0,M1) = R1;
```

Certain sequences of instructions cause incorrect results on the DSP and are flagged as errors by the DSP 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 the *ADSP-21160 SHARC DSP Instruction Set 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 DSP's universal registers, see Table A-1 on page A-2.
- *`Dreg`* indicates any data register; for a list of the DSP's data registers, see the Data Register File registers listed in Table A-1 on page A-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-21262 processor contains a large, dual-ported internal memory for single cycle, simultaneous, independent accesses by the core processor and I/O processor. The dual-ported memory, in combination with three separate on-chip buses, allow two data transfers from the core and one transfer from the I/O processor in a single cycle. Using the I/O bus, the I/O processor provides data transfers between internal memory and the DSP's communication ports (serial ports and parallel port) without hindering the DSP core's access to memory. This chapter describes the DSP's memory and how to use it.

The DSP provides access to 8- and 16-bit external memory through the DSP's parallel port. External memory is only accessible via DMA (direct memory access). For information on connecting and timing accesses to external memory devices, see the product specific peripherals manual.

The DSP contains up to 2M bits of internal RAM and up to 4M bits of internal ROM depending on the specific part number[1]. Regardless, each block can be configured for different combinations of code and data storage. All of the memory can be accessed as 16-bit, 32-bit, 48-bit, or 64-bit words. The DSP 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,

[1] For specific memory information, see your ADSP-2126x product specific data sheet.

(typically block 1) for transfers, and the other block (typically block 0) stores instructions and data using the PM bus. Using the DM bus and PM bus 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.

Internal Memory

The ADSP-21262 and ADSP-21266 SHARC DSPs contain 2M bits of internal RAM and 4M bits of internal ROM. Block 0 has 1M bit RAM and 2M bits ROM. Block 1 has 1M bit RAM and 2M bits ROM. Table 5-1 shows the maximum number of data or instruction words that can fit in each internal memory block.

The ADSP-21262 processor family members are available with varying amounts of internal ROM and RAM. For a complete list, visit our web site at www.analog.com\SHARC.

Table 5-1. Words Per Internal Memory Block (ADSP-21262/21266 Models)

Word Type	Bits Per Word	Maximum Number of Words in Block 0		Maximum Number of Words in Block 1	
		1M bit RAM	2M bits ROM	1M bit RAM	2M bits ROM
Instruction	48 bits	21.33K words	42K words	21K words	42K words
Long Word Data	64 bits	16K words	32K words	16K words	32K words
Extended-Precision Normal Word Data	40 bits	21K words	42K words	21K words	42K words

Table 5-1. Words Per Internal Memory Block (ADSP-21262/21266 Models) (Cont'd)

Word Type	Bits Per Word	Maximum Number of Words in Block 0		Maximum Number of Words in Block 1	
		1M bit RAM	2M bits ROM	1M bit RAM	2M bits ROM
Normal Word Data	32 bits	32K words	64K words	32K words	64K words
Short Word Data	16 bits	64K words	128K words	64K words	128K words

DSP 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 DSPs require greater data throughput than the Von Neumann architecture provides, many DSPs use memory architectures that have separate data and address buses for program and data storage. These two sets of buses let the DSP retrieve a data word and an instruction simultaneously. This type of memory architecture is called Harvard architecture.

SHARC DSPs 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.

DSP 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 DSP core and I/O processor in every cycle—no extra cycles are incurred if the DSP core and the I/O processor access the same block.

A memory access conflict can occur when the processor core attempts two accesses to the same internal memory block in the same cycle. When this conflict, known as a block conflict occurs, an extra cycle is incurred. The DM bus access completes first and the PM bus access completes in the following (extra) cycle.

For more information on how the buses access memory blocks, see "Internal Memory" on page 5-2.

Buses

As shown in Figure 5-1, the DSP has three sets of internal buses connected to its dual-ported memory, the Program Memory (PM), Data Memory (DM), and I/O Processor (I/O) buses. The PM and DM buses share one memory port and the I/O bus connects to the other port. Memory accesses from the DSP's core (computational units, data address generators, or program sequencer) use the PM or DM buses, while the I/O processor uses the I/O bus for memory accesses. The I/O processor's parallel port (PP) bus can access external memory devices. For more information about the external memory and I/O capabilities of the processor, see the processor specific peripherals manual.

Internal Address and Data Buses

Figure 5-1 on page 5-6 shows that the PM and DM buses have access to internal memory.

The DSP's DM and PM buses can access internal memory independently. The I/O processor can perform DMA between external and internal memory without conflicts with the DM and PM buses.

Addresses for the PM and DM buses come from the DSP'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 DSP'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.

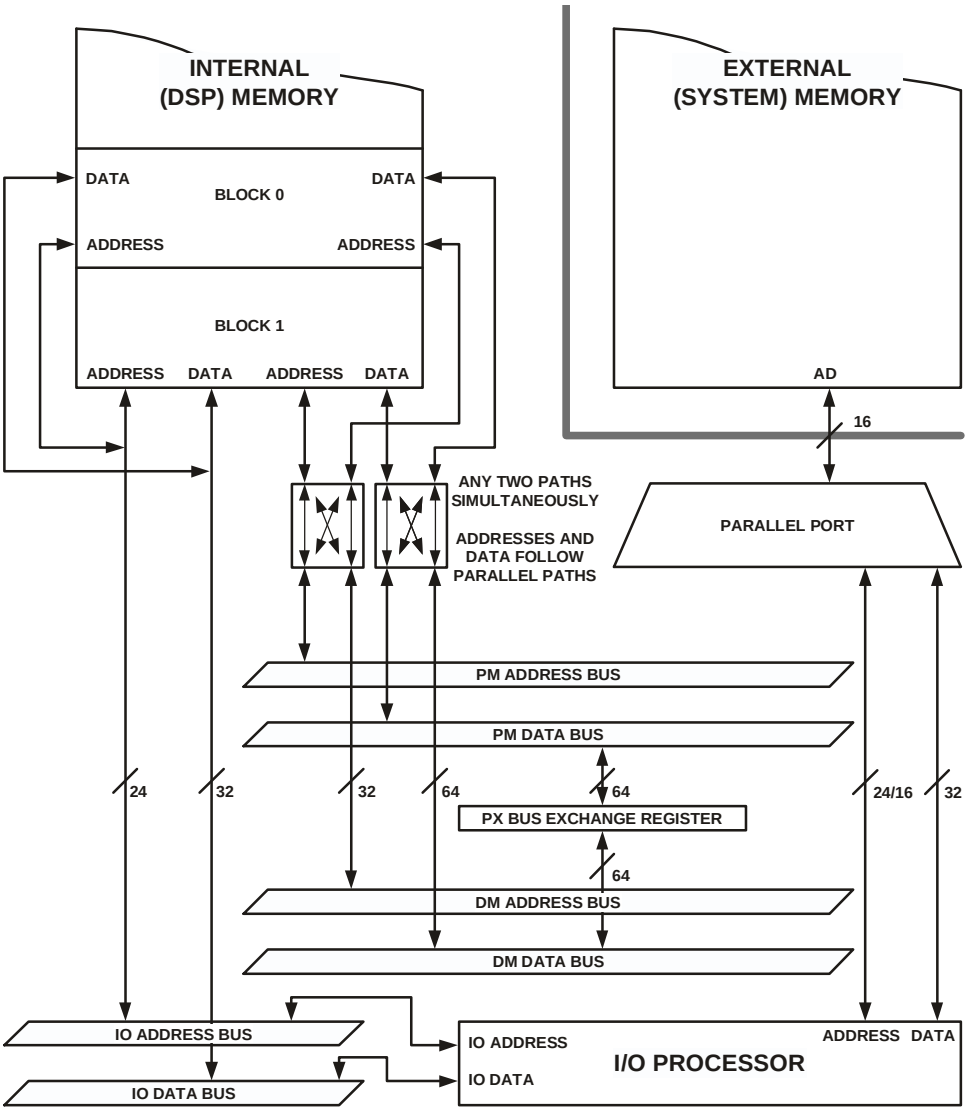


Figure 5-1. ADSP-21262 Processor Memory and Internal Buses Block Diagram

Because the DSP's internal memory is arranged in four 16-bit wide by 96K 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 the how the DSP works with memory words, see "Memory Organization and Word Size" on page 5-15.

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 DSP to any other register or to any internal memory location in a single cycle. As shown in Figure 5-2, 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 alignment of `PX1` and `PX2` within `PX` appears in Figure 5-2.

The `PX1`, `PX2`, and the combined `PX` registers are Universal registers (`Ureg`) 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 5-2 shows the bit alignment and gives an example of instructions for register-to-register transfers.

Instruction Examples

```
PX = DM(0x80000)(LW);
PX = DM(0x40000);
```

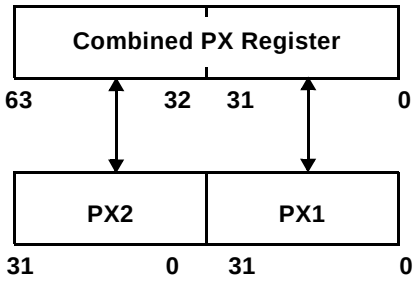


Figure 5-2. PM Bus Exchange (PX, PX1, and PX2) Registers

Figure 5-2 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.

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 5-5 shows these transfers.

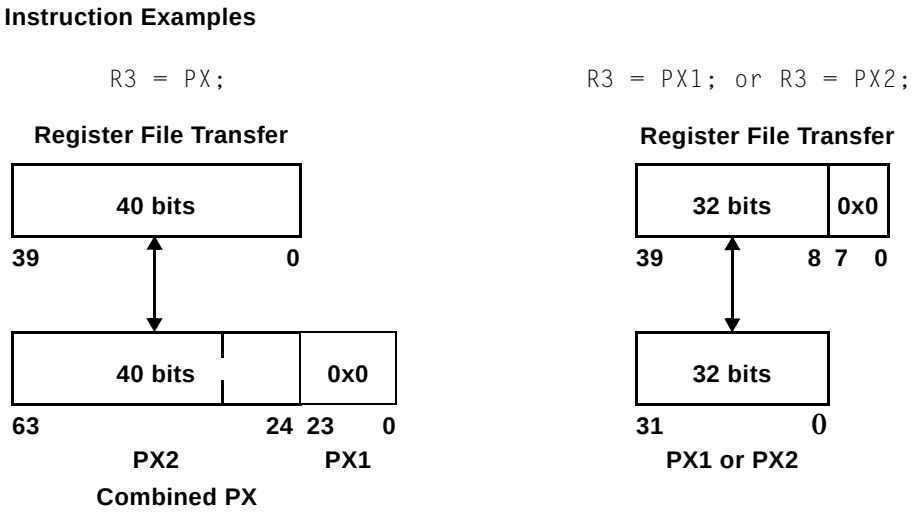


Figure 5-3. PX, PX1, and PX2 Register-to-Register Transfers

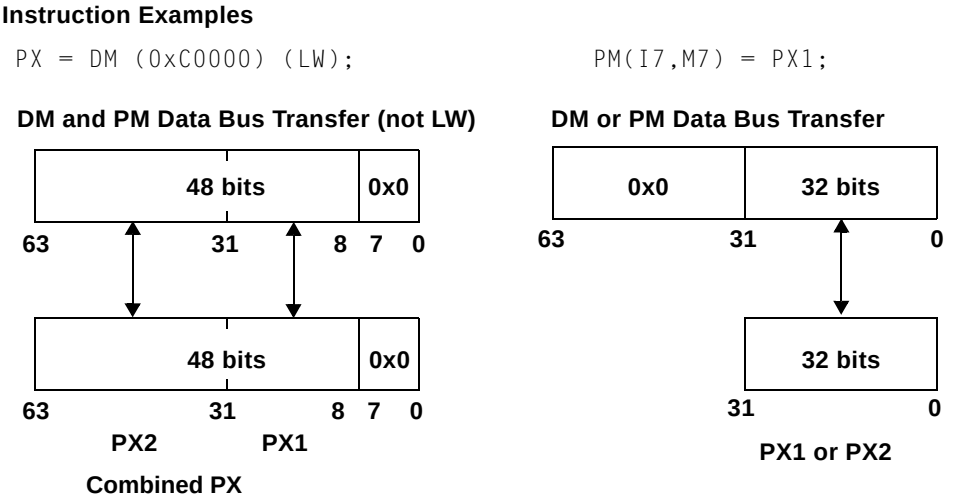


Figure 5-4. PX, PX1, PX2 Register-to-Memory Transfers on DM (LW) or PM (LW) Data Bus

Figure 5-5 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 8 bits.

The status of the memory block's Internal Memory Data Width (IMDWX) 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 on page 5-9.

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 (0x80000)LW;
```

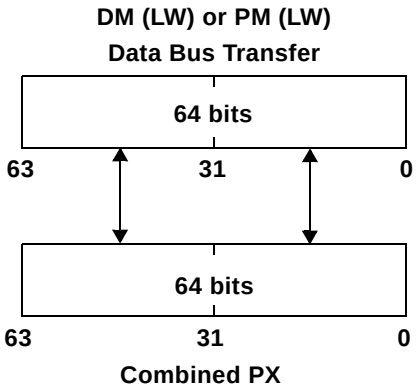


Figure 5-5. PX Register-to-Memory Transfers on PM Data Bus

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 forced to be 64-bit (four column) transfers with the LW (long word) mnemonic.

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 */
```

Instruction Example

```
PX = USTAT1;
```

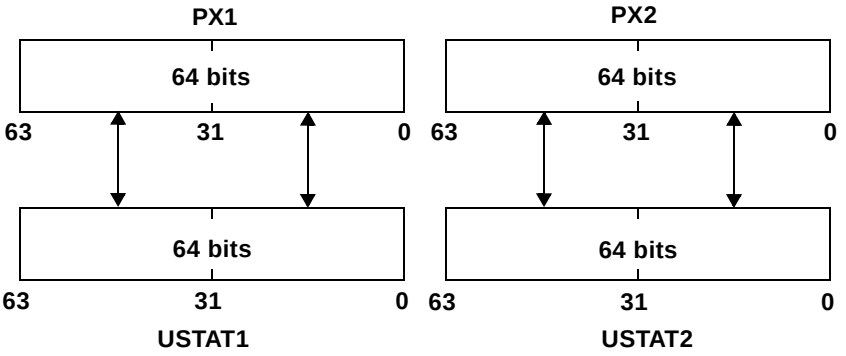


Figure 5-6. PX Register-to-Internal Memory Transfers Over the PM or DM DATA Bus

ADSP-21262 Processor Memory Map

The ADSP-21262 processor family is composed of a variety of models that have varying amounts of RAM and/or ROM memory. However, in general, there are two memory spaces: internal memory space and external (DMA) memory space. These spaces have these definitions:

- **Internal memory space.** This space ranges from address 0x00 0000 through 0x1F FFFF. Internal memory space refers to the DSP's on-chip RAM, on-chip ROM, and memory-mapped registers.
- **External (DMA) memory**. For information on external DMA memory space please refer to the product specific data sheet.

The ADSP-21262 processor has two blocks of RAM that contain up to 1M bit of memory each, and two blocks of ROM that contain up to 2M bits of memory each. Each block is physically comprised of four 16-bit columns. "Wrapping", as shown in Figure 5-8 on page 5-17, allows the memory to efficiently store 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.

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 (ADSP-21262 Example)

```
#include <def2126x.h>
```

```
DM(0x00040000) = PX;    /* long word transfer
                           (64 bits/four columns) */

DM(0x00080000) = R0;    /* normal word transfer
                           (32 bits/two columns) */

DM(0x00100000) = 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(0x00080000) = 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).

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 two bits in the SYSCTL register, IMDW0 and IMDW1, which determine whether access to each block is 32 or 40 bits. For more information, see “Accessing Memory” on page 5-27.

The I/O processor’s memory-mapped registers control the system configuration of the DSP and I/O operations. For information about the I/O Processor, see the product specific peripherals manual. 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.

Memory Organization and Word Size

The DSP'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 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 Words 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 DSP 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.

For long word (64 bits), normal word (32 bits), and short word (16 bits) memory accesses, the DSP selects from fixed columns in memory. No rotations of words within columns occur for these data types.

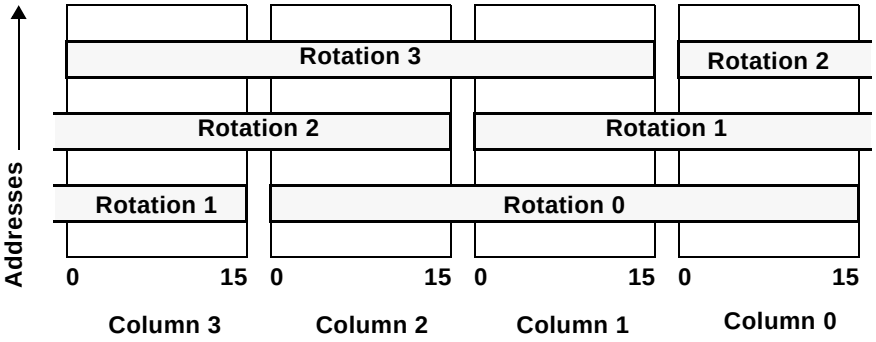


Figure 5-7. 48-Bit Word Rotations

Figure 5-8 shows the memory ranges for each data size in the DSP's internal memory.

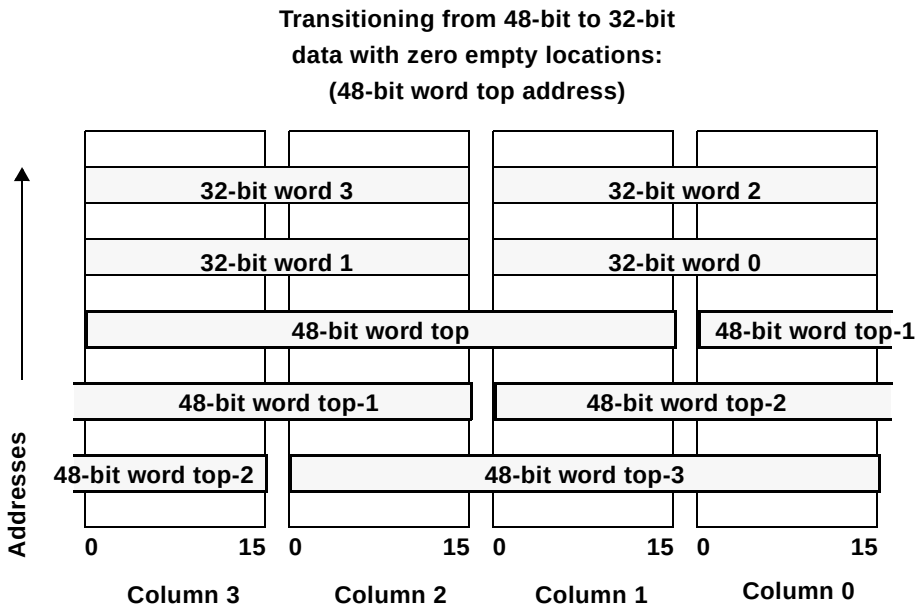


Figure 5-8. Mixed Instructions and Data With No Unused Locations

Mixing 32-Bit Words and 48-Bit Words

The DSP's memory organization lets programs freely place memory words of all sizes (see "Memory Organization and Word Size" on page 5-15) with few restrictions (see "Restrictions on Mixing 32-Bit Words and 48-Bit Words" on page 5-19). 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 DSP's memory.

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.

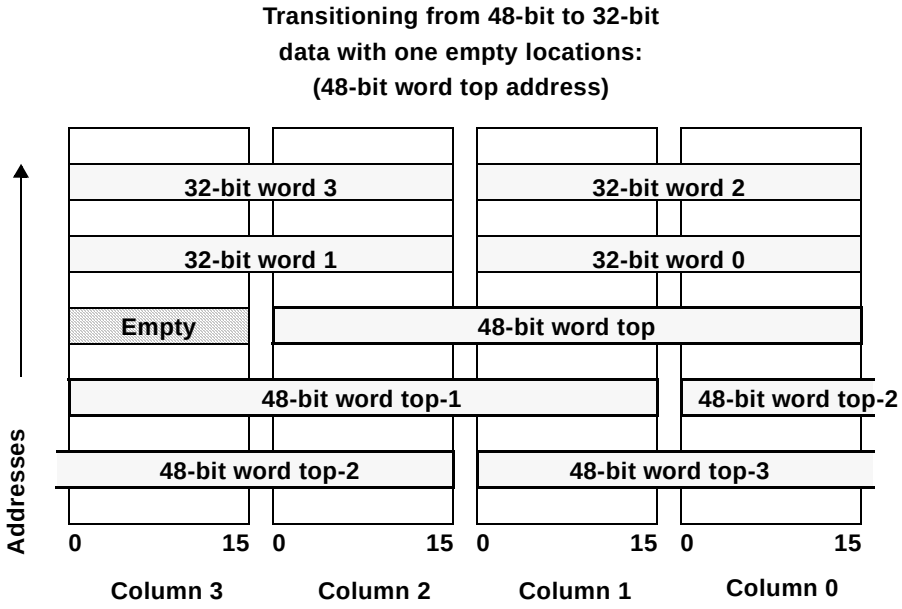


Figure 5-9. Mixed Instructions and Data With One Unused Location

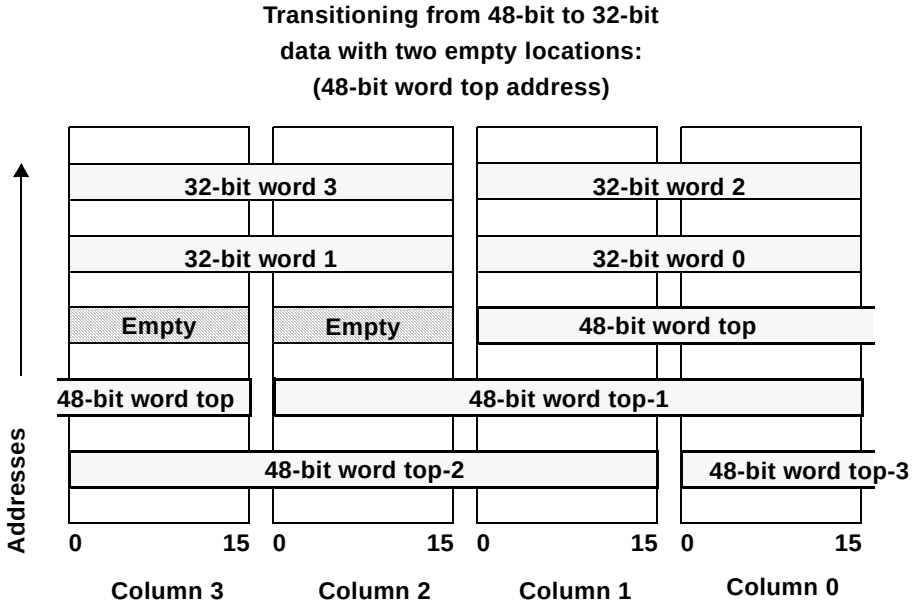


Figure 5-10. Mixed Instructions and Data With Two Unused Locations

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 four column data (32-bit words) in memory. These restrictions apply to mixing 48 and 32-bit words, because the DSP 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 four column data that minimizes the gap after three column data is useful for programs that are mixing three and four 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 shown in Listing 5-2.

Listing 5-2. Starting Address

```
m = B + 2 [(n MOD K) - TRUNC (n MOD K) / 4)]
```

where:

- **K** is 21844 for RAM and 43690 for ROM
- **n** is the number of contiguous 48-bit words allocated in the internal memory block (n < 43,690 for ROM, n < 21844 for RAM)
- **B** is the base normal word address of the internal memory block; if **B** = 0x80000 (Block 0) else **B** = 0xC0000 (Block 1)
- **m** is the first 32-bit normal word address to use after the end of 48-bit words

Example: Calculating a Starting Address for 32-Bit Addresses

The last valid address is 0x82694. The number of 48-bit words (n) is:

```
n = 0x82694 - 0x80000 + 1 = 0x2695
```

When you convert 0x2695 to decimal representation, the result is 9877.

The base (B) normal word address of the internal memory block is 0x80000 since the condition `0 < 10922` is TRUE.

The first 32-bit normal word address to use after the end of the 48-bit words is given by:

```
m = 0x80000 + 2 [(9877 MOD 21844)- TRUNC (9877 MOD 21844)/4]

m = 0x80000 + 14816decimal
```

Convert to a hexadecimal address:

```
14816decimal = 0x39E0

m = 0x80000 + 0x39E0 = 0x839E0
```

The first valid starting 32-bit address is 0x839E0. The starting address must begin on an even address.

48-Bit Word Allocation

Another useful calculation for programs that are mixing three and four 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 four column (32-bit) data, the number of 48-bit words that most efficiently uses memory can be determined as shown in Listing 5-3.

Listing 5-3. 48-Bit Word Allocation

```
n = TRUNC{4[(m - B) / 2] / 3]} + B
```

where:

- **m** is the first 32-bit normal word address after the end of 48-bit words (1m values falls in the valid normal word address space)
- **B** is the base normal word address of the internal memory block; **B** = 0x80000 (block 0) else **B** = 0xC0000 (block 1) for valid m values
- **n** is the number of contiguous 48-bit words the system allocates in the internal memory block

Using Boot Memory

As shown in Figure 5-10, the DSP supports an external boot EPROM via the parallel port. The boot EPROM provides one of the methods for automatically loading a program in to the internal memory of the DSP after power-up or after a software reset. For information about boot options and the booting process, see the product specific peripherals manual.

Reading From Boot Memory

When the DSP boots from an EPROM, the DSP's I/O processor is hard-wired to load 256 instructions automatically from EPROM (via DMA). Once the initial 256-word DMA is complete, the DSP typically needs to maintain access to boot memory.

Internal Interrupt Vector Table

The default location of the ADSP-21262 processor's interrupt vector table (IVT) depends on the DSP's booting mode. When the processor boots from an external source (EPROM, SPI port master or slave booting), the vector table starts at address 0x0008 0000 (normal word). When the processor is in "no boot" mode (runs from internal ROM location 0x000A 0000 without loading), the interrupt vector table starts at address 0x000A–0000.

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 table starts at address 0x0008 0000 (internal memory) regardless of the booting mode.

Internal Memory Data Width

The DSP'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; only writing 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, only reading the 32 most significant bits (MSBs).

The Program 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 5-19.

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 DSP is in Single-Instruction, Multiple-Data (SIMD) mode. In SIMD mode, many data access operations differ from the DSP'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 5-24. For more information on complementary (implicit) memory locations in SIMD mode accesses, see "Accessing Memory" on page 5-27.

Broadcast Register Loads

The DSP's BDCST1 and BDCST9 bits in the MODE1 register control broadcast register loading. When broadcast loading is enabled, the DSP 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 DSP 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 5-2 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 5-27.

Table 5-2. 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 DSP 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 A-11.

Unaligned 64-Bit Memory Access

The DSP 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 A-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(0x80200) = r0(lw);      /* even address in 32-bit, access
                              is aligned */
pm(0x80201) = 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 5-25 and "Unaligned 64-Bit Memory Access" on page 5-26, the DSP can provide illegal access information for long word or I/O register accesses. When these conditions occur, the DSP 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 DSP 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

The DSP 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 5-23. 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 5-19.

The DSP'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
- Mode of DSP—SISD, SIMD, or broadcast load

Access Word Size

The DSP'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 the processor model data sheet.

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).

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(0x80000) = R0 (LW);
/* The data in R0 moves to location DM(0x80000), and the data in
R1 moves to location DM(0x80001) */

R0 = DM(0x80003)(LW);
/* The data at location DM(0x80002) moves to R0, and the data at
location DM(0x80003) moves to R1 */
```

The example shows that R0 and R1 are neighbor registers in the same processing element. Table 5-3 lists the other neighbor register assignments that apply to long word accesses.

In unforced long word accesses (accesses to LW memory space), the DSP 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 5-26.

The Long word (LW) mnemonic only effects normal word address accesses and overrides all other factors (PEYEN, IMDWx).

Table 5-3. 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 and Extended-Precision Normal Word 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 Figure 5-8 on page 5-17. For more information on configuring memory for extended-precision normal word accesses, see "Internal Memory Data Width" on page 5-23.

The DSP 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 DSP memory spaces. The address ranges for memory accesses appear in Figure 5-8 on page 5-17, Figure 5-10 on page 5-19, and Figure 5-11 on page 5-37.

The register file source or destination of a normal word access is a single 40-bit data register. The DSP 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 Figure 5-8 on page 5-17.

The register file source or destination of such an access is a single 40-bit data register. The DSP zero-fills the least significant 8 bits on loads and truncates these bits on stores. Depending on the value of the SSE bit in the MODE1 system register, the DSP loads the register's upper 16 bits 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 DSP's memory. The SYSCTL register is described in the *ADSP-21262/21266 SHARC DSP Peripherals Manual*, Table A-2 on page A-5 lists all the bits in the MODE1 register, and Table A-3 on page A-12 lists all the bits in the MODE2 register.

SYSCTL Register Control Bits

The SYSCTL register is described in the *ADSP-21262/21266 SHARC DSP Peripherals Manual*. 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 0x0008 0000 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 10-9 (IMDWx) selects the normal word data access size for internal memory Block 0 and Block1. 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 in SIMD mode, (if 1) or disables PEy in 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 5-34. and "Secondary Processing Element (PEy)" on page 2-44.

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 5-34. For more information on broadcast load mode, see "Broadcast Register Loads" on page 5-24.

Single- and Dual-Data Accesses

The number of transfers that occur in a cycle influences the data access operation. As described in "DSP Architecture" on page 5-3, the DSP 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

Table 5-4 lists the DSP's possible memory transfer modes and provides a cross-reference to examples of each memory access option that stems from the DSP's data access options.

Table 5-4 shows the transfer modes that stem from the following data access options:

- The mode of the DSP: 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: single or dual-data

To take advantage of the DSP'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 the *ADSP-21160 SHARC DSP Instruction Set 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.

Table 5-4. Memory Transfer Modes Cross Reference

Access Type	DSP Mode	Address Space			
		Long Word	Extended-Precision	Normal Word	Short Word
Single- Data Access	SISD mode	LW on page 5-58	EW on page 5-50	NW on page 5-44	SW on page 5-36
	SIMD mode	LW on page 5-58	EW on page 5-56	LW on page 5-46	SWx2 on page 5-38
	B-cast Load	LW Figure 5-22	EW Figure 5-31	NW Figure 5-29	SW Figure 5-27
Dual-Data Access	SISD mode	LW on page 5-60	EW on page 5-54	NW on page 5-48	SW on page 5-60
	SIMD mode	LW on page 5-62	EW on page 5-60	LW on page 5-62	SWx2 on page 5-42
	B-cast Load	LW Figure 5-34	EW Figure 5-22	NW Figure 5-30	SW Figure 5-28
Symbols: LW = 64-bit data value (two 32-bit values), EW = 40-bit data value (48-bit value), NW = 32-bit data value, SW = 16-bit data value, and SWx2 = two 16-bit data values					

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 DSP 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 DSP 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 targets a PEy register using the syntax SX.

The cross (†) in the PEx registers in Figure 5-11 indicates that the DSP 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.

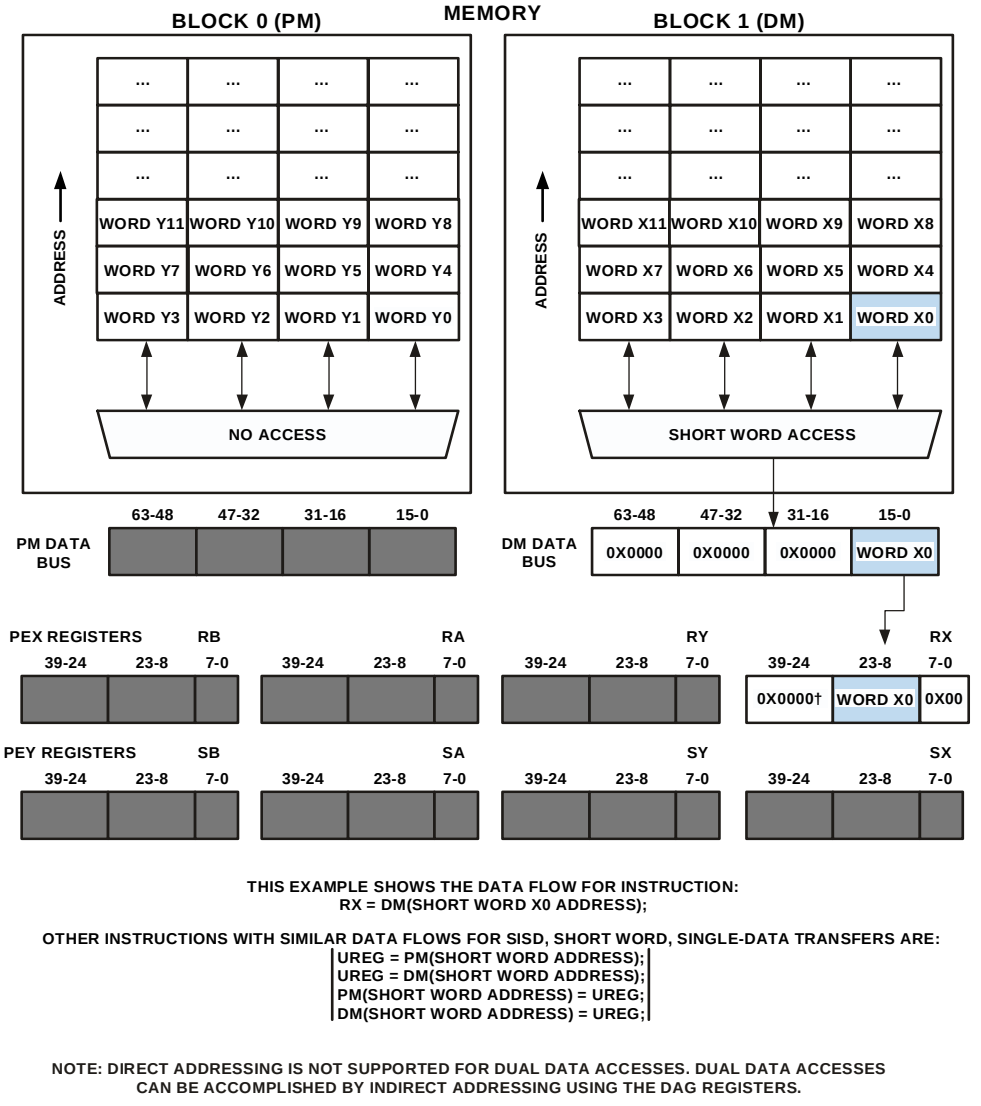


Figure 5-11. Short Word Addressing of Single-Data in SISD Mode

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 DSP 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 DSP drives the other short word lanes of the PM or DM data buses with zeros.

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 DSP uses that register's complement RX as the implicit target. For more information on complementary registers, see "Secondary Processing Element (PEy)" on page 2-44.

The cross (†) in the PEx and PEy registers in Figure 5-12 indicates that the DSP 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 DSP accesses short words sequentially in memory. Table 5-5 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 5-75.

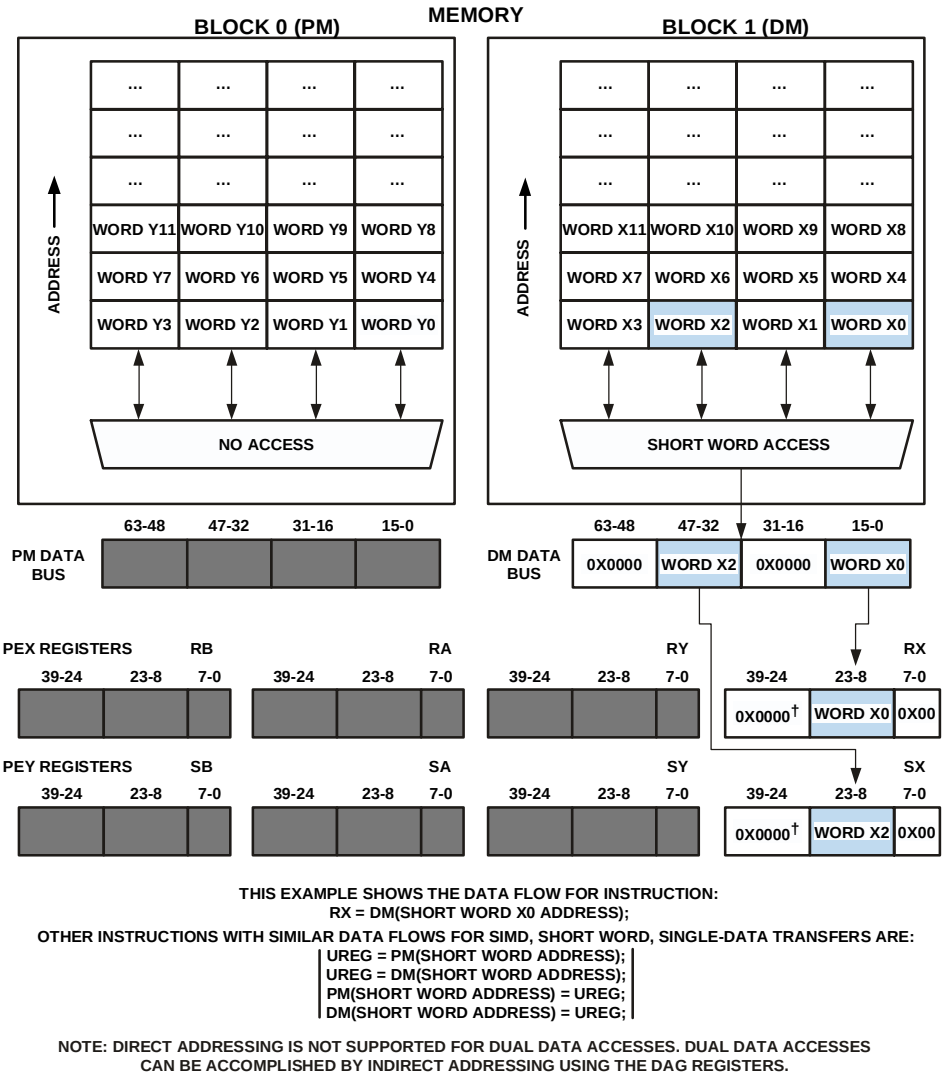


Figure 5-12. Short Word Addressing of Single-Data in SIMD Mode

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 DSP 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 DSP 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 5-64.

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 DSP 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.

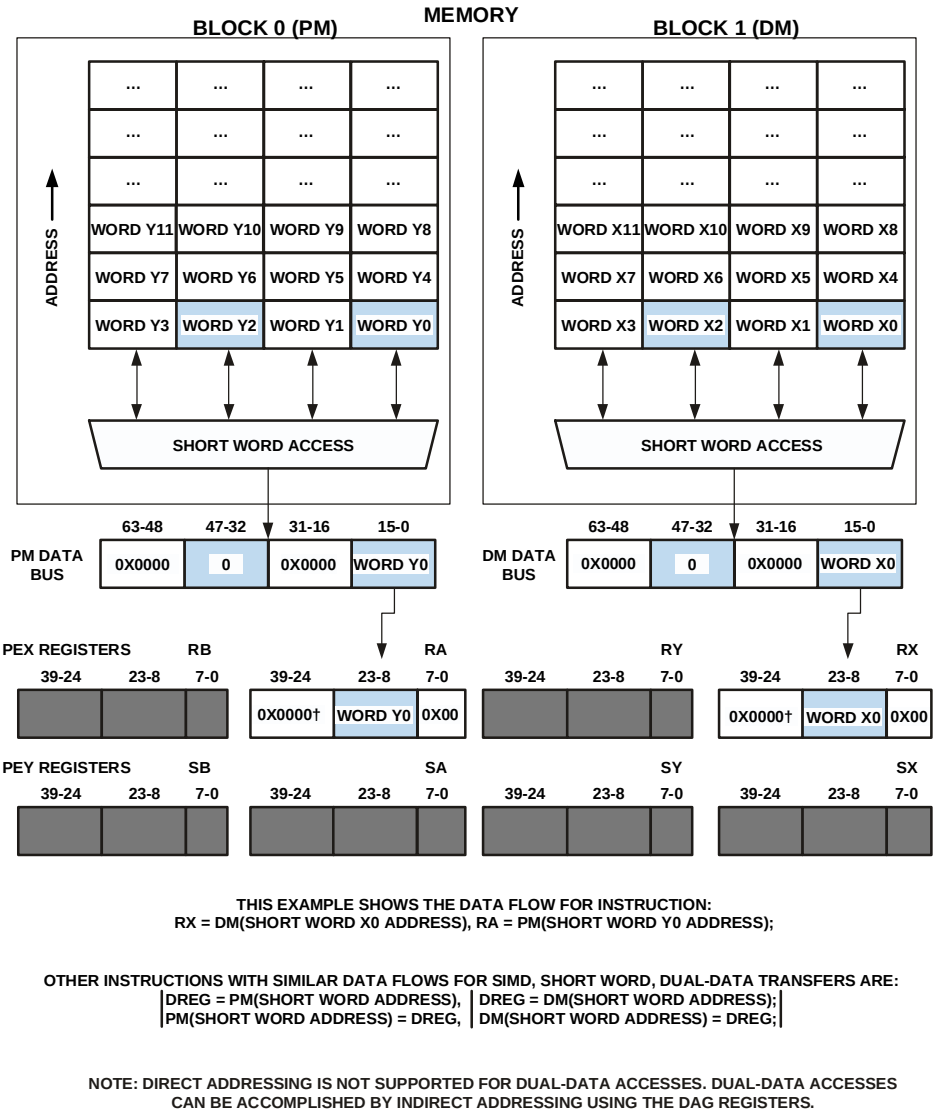


Figure 5-13. Short Word Addressing of Dual-Data in SISD Mode

Short Word Addressing of Dual-Data in SIMD Mode

Figure 5-14 shows the SIMD, dual-data, short word addressed access mode. For short word addressing, the DSP treats the data buses as four 16-bit short word lanes. The explicitly addressed (named in the instruction) 16-bit values are transferred using the least significant short word lanes of the PM and DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) short word values are transferred using the 47-32 bit short word lanes of the PM and DM data buses. The DSP 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 5-64.

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 DSP uses those registers' complements, `RX` and `RA`, as the implicit targets. For more information on complementary registers, see "Secondary Processing Element (PEy)" on page 2-44.

The cross (†) in the `PEx` and `PEy` registers in Figure 5-14 indicates that the DSP 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.

Figure 5-14 shows the data path for one transfer. 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 5-76.

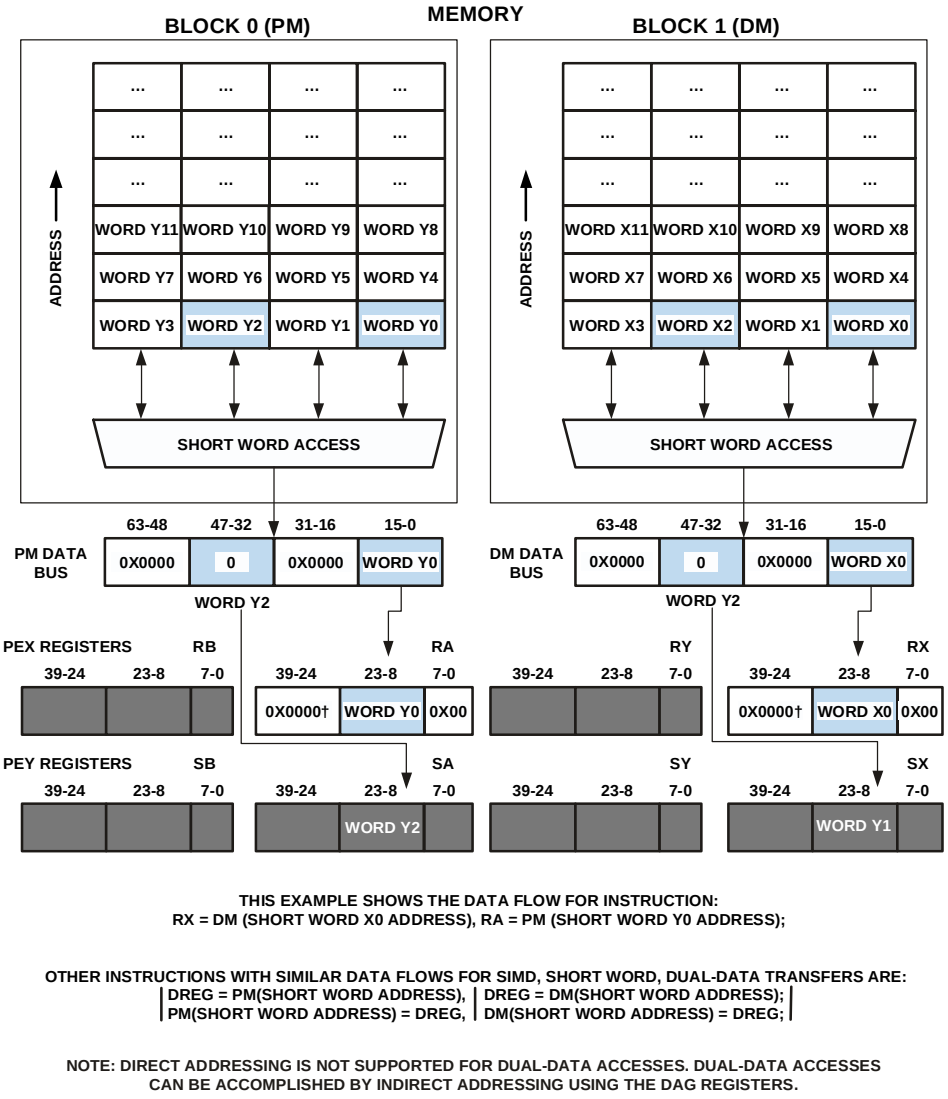


Figure 5-14. Short Word Addressing of Dual-Data in SIMD Mode

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 DSP 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 DSP 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 DSP zero-fills the least significant 8 bits of the data register on loads and truncates these bits on stores to memory.

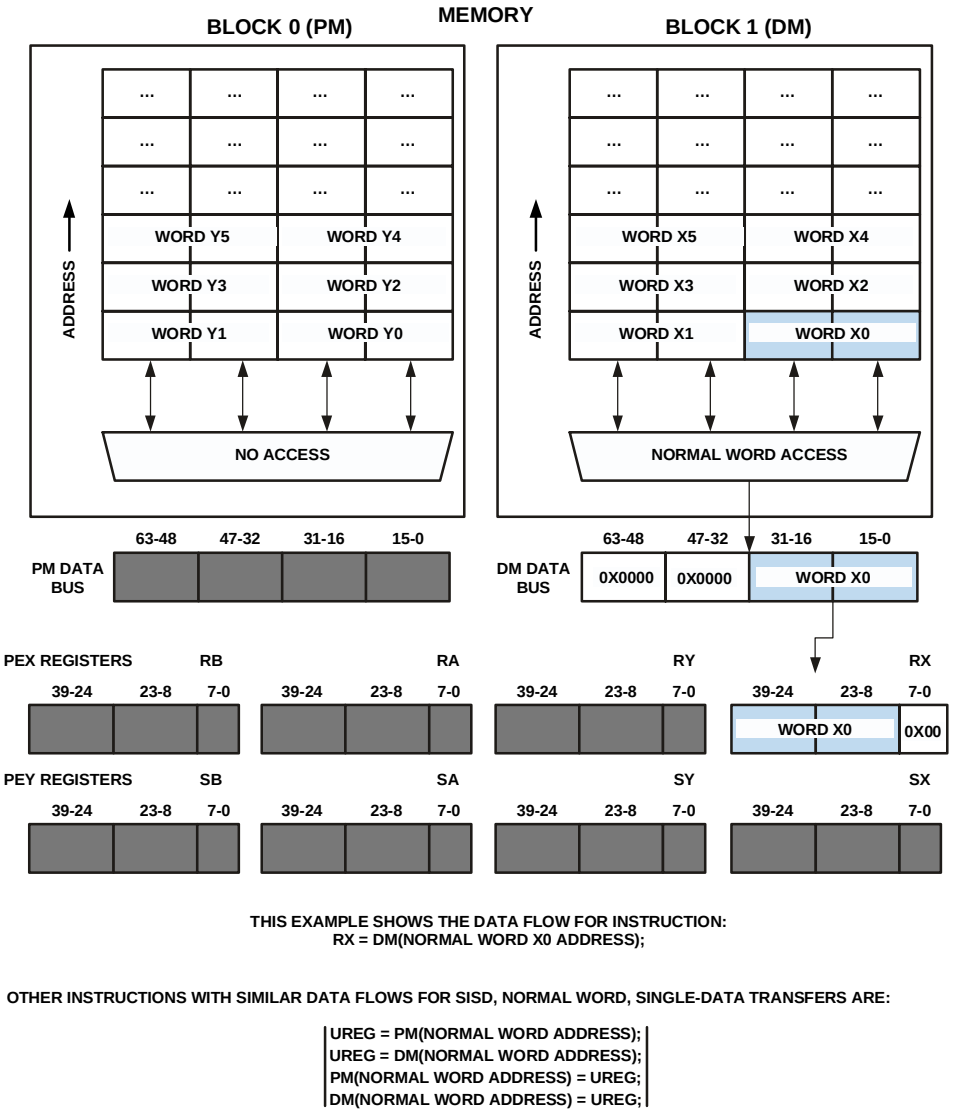


Figure 5-15. Normal Word Addressing of Single-Data in SISD Mode

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 DSP 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 DSP would use that register's complement, `RX`, as the implicit target. For more information on complementary registers, see "Secondary Processing Element (PEy)" on page 2-44.

For normal word accesses, the DSP 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 DSP accesses normal words sequentially in memory (see Table 5-5). For more information on arranging data in memory to take advantage of this access pattern, see Figure 5-34 on page 5-76.

Table 5-5. Normal Word Addressing in SIMD Mode

Implicit Normal Word Accessed	Explicit Normal Word Accessed
Word X0 (address LSB = 0)	Word X1 (address LSB = 1)
Word X1 (address LSB = 1)	Word X2 (address LSB = 0)

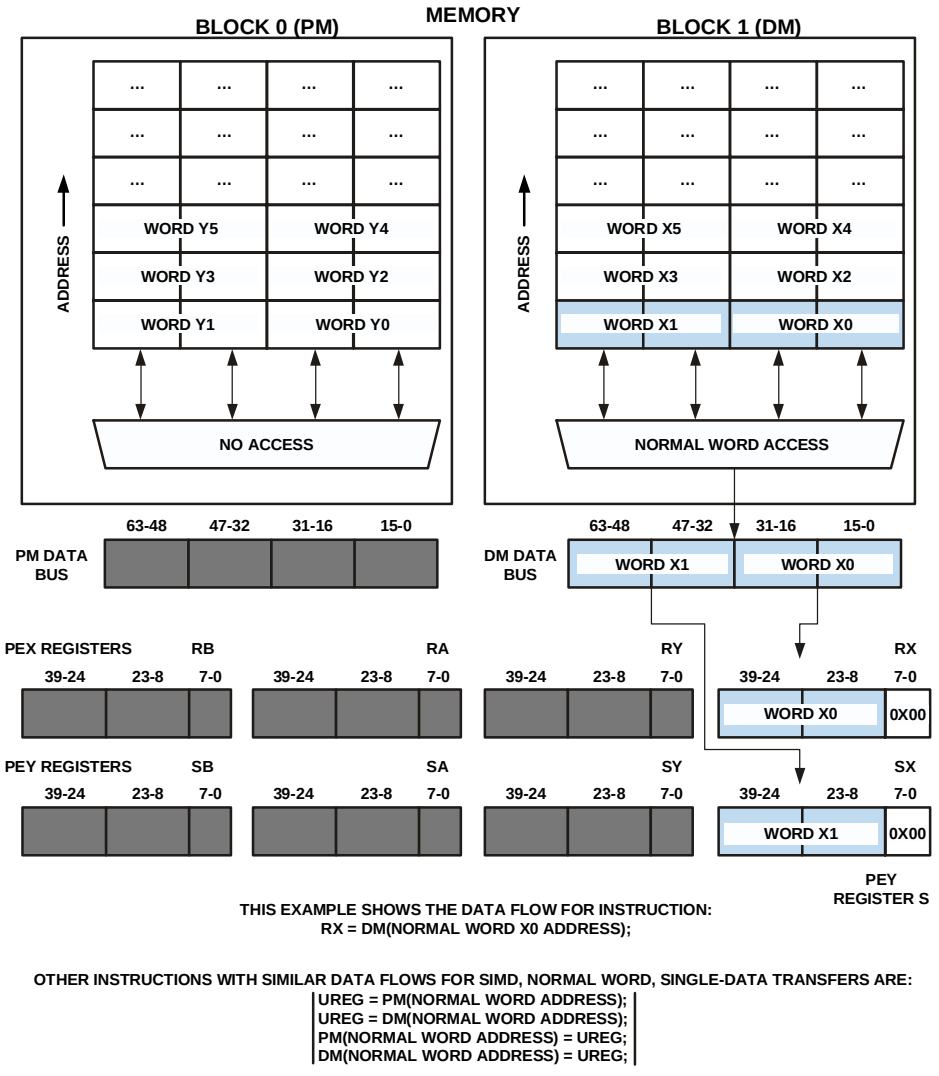


Figure 5-16. Normal Word Addressing of Single-Data in SIMD Mode

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 DSP 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 DSP 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 5-64.

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 DSP zero-fills the least significant 8 bits of the data register on loads and truncates these bits on stores to memory.

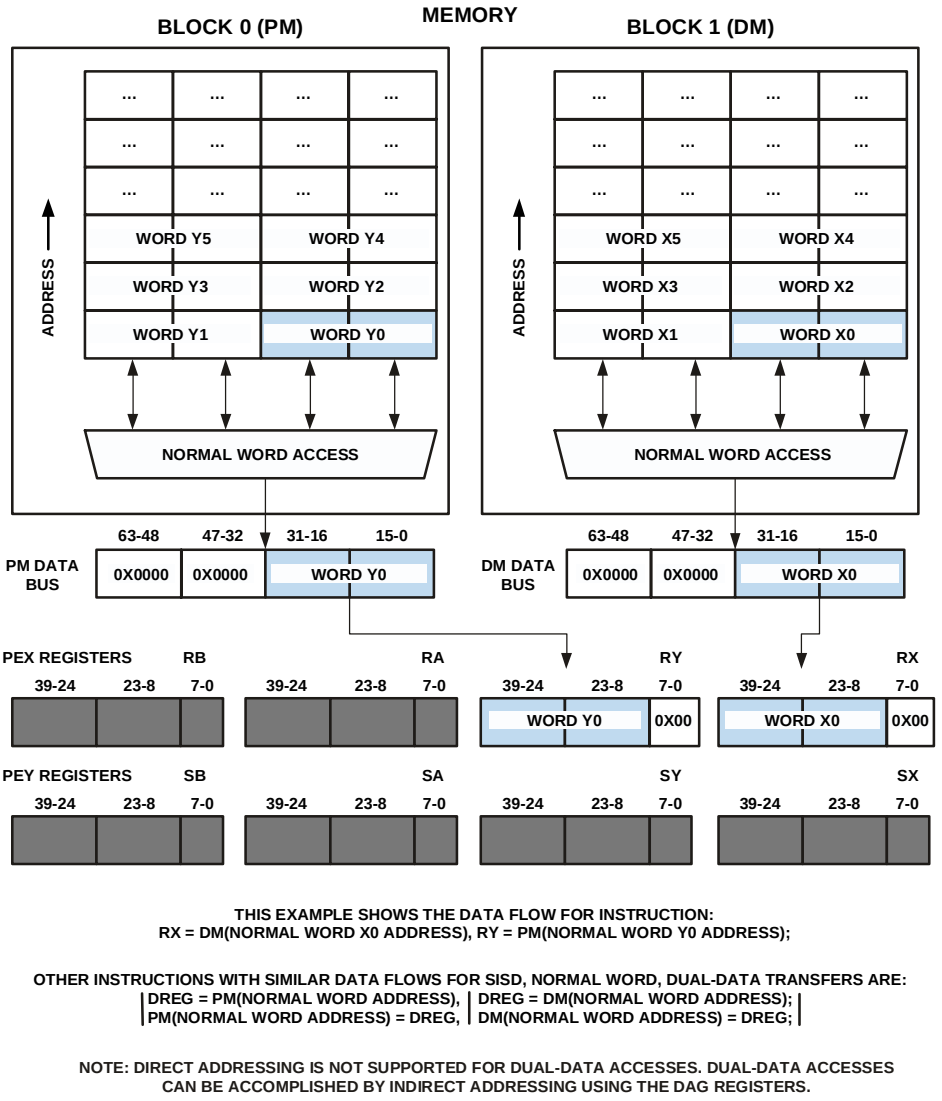


Figure 5-17. Normal Word Addressing of Dual-Data in SISD Mode

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 DSP 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 5-64.

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 DSP 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 2-44.

For normal word accesses, the DSP 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 DSP accesses normal words sequentially in memory as shown in Table 5-5 on page 5-46. For more information on arranging data in memory to take advantage of this access pattern, see Figure 5-34 on page 5-76.

Extended-Precision Normal Word Addressing of Single-Data

Figure 5-19 on page 5-53 displays a possible single-data, 40-bit extended-precision normal word addressed access. For extended-precision normal word addressing, the DSP treats each data bus as a 40-bit

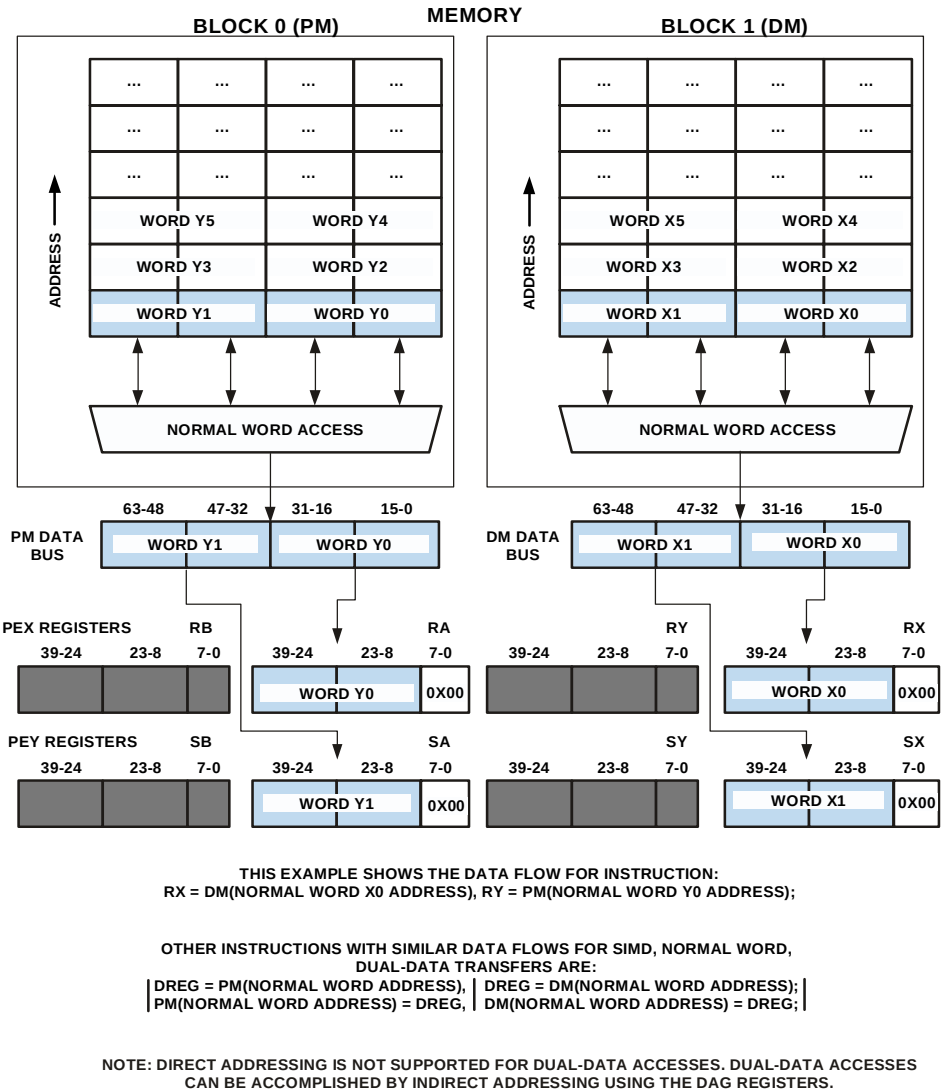


Figure 5-18. Normal Word Addressing of Dual-Data in SIMD Mode

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 DSP 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.

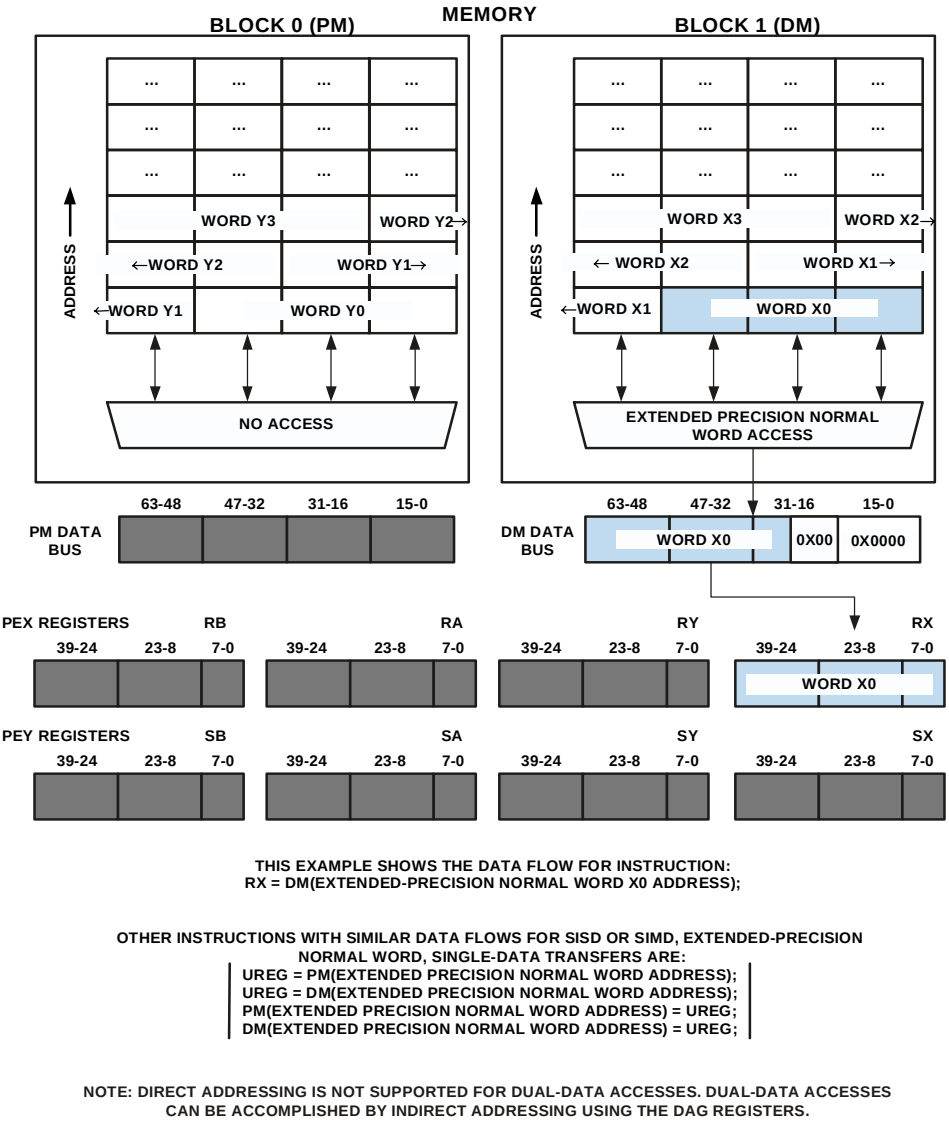


Figure 5-19. Extended-Precision Normal Word Addressing of Single-Data

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 DSP 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 DSP 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 5-64.

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`.

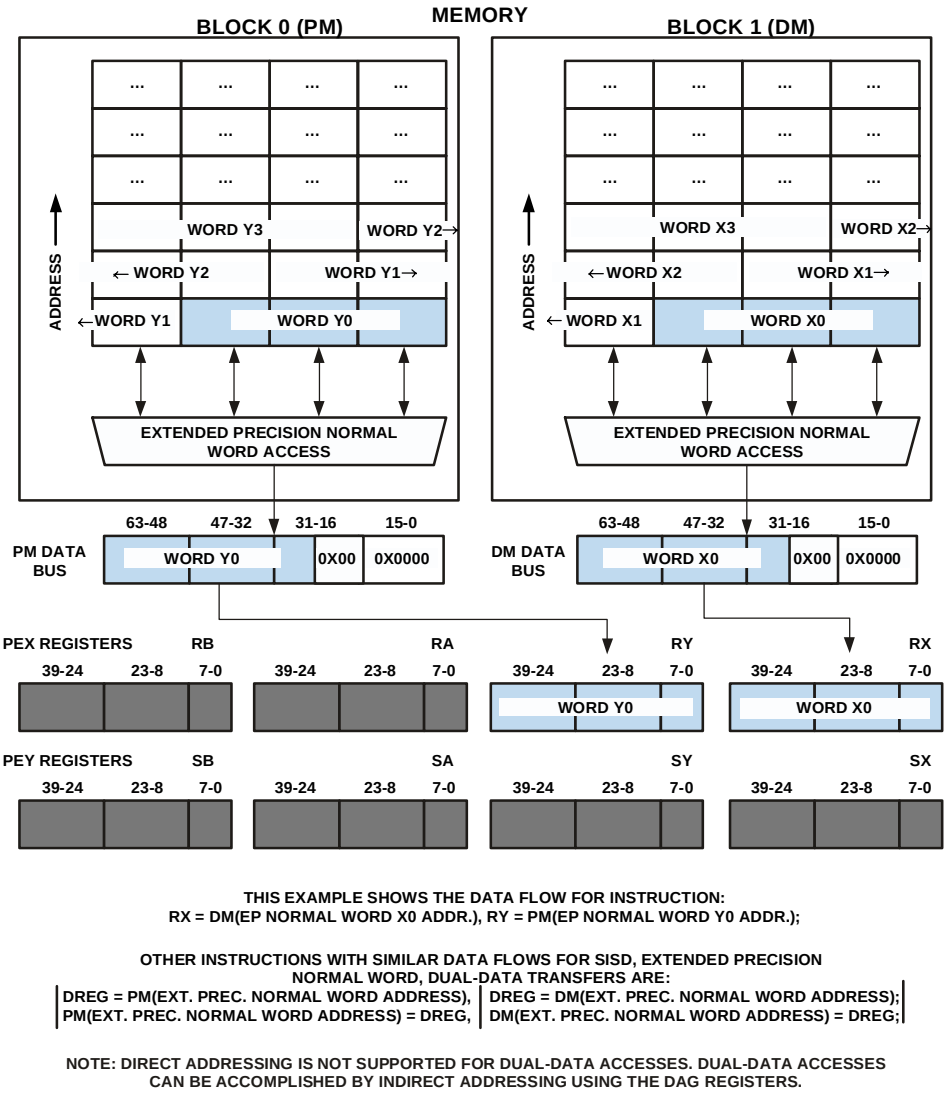


Figure 5-20. Extended-Precision Normal Word Addressing of Dual-Data in SISD Mode

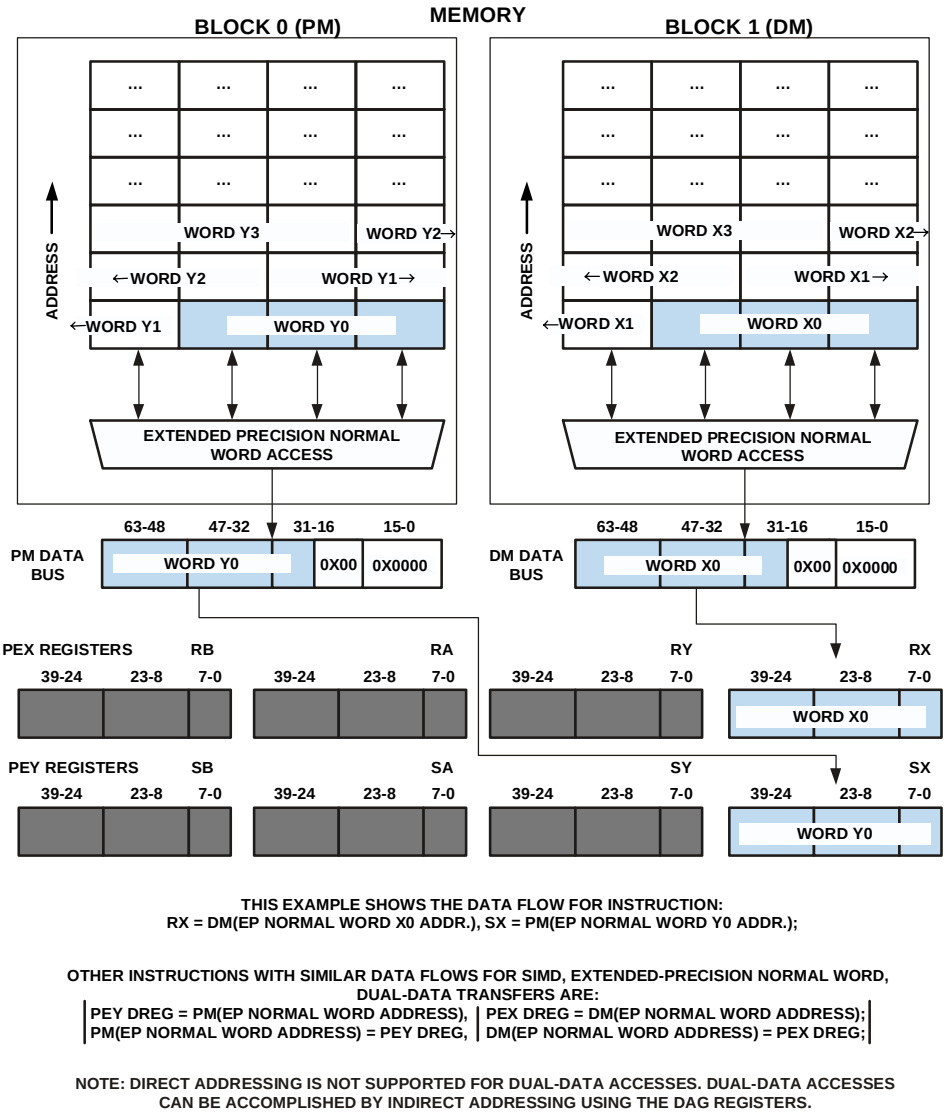
Extended-Precision Normal Word Addressing of Dual-Data in SIMD Mode

Figure 5-21 shows the SIMD, dual-data, 40-bit extended-precision normal word addressed access mode. For extended-precision normal word addressing, the DSP 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 DSP 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 5-64.

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 DSP 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 example targets `PEy` registers when using the syntax `SX`. For more information on how neighbor registers (listed in Table 5-3) work, see "Long Word (64-Bit) Accesses" on page 5-28.

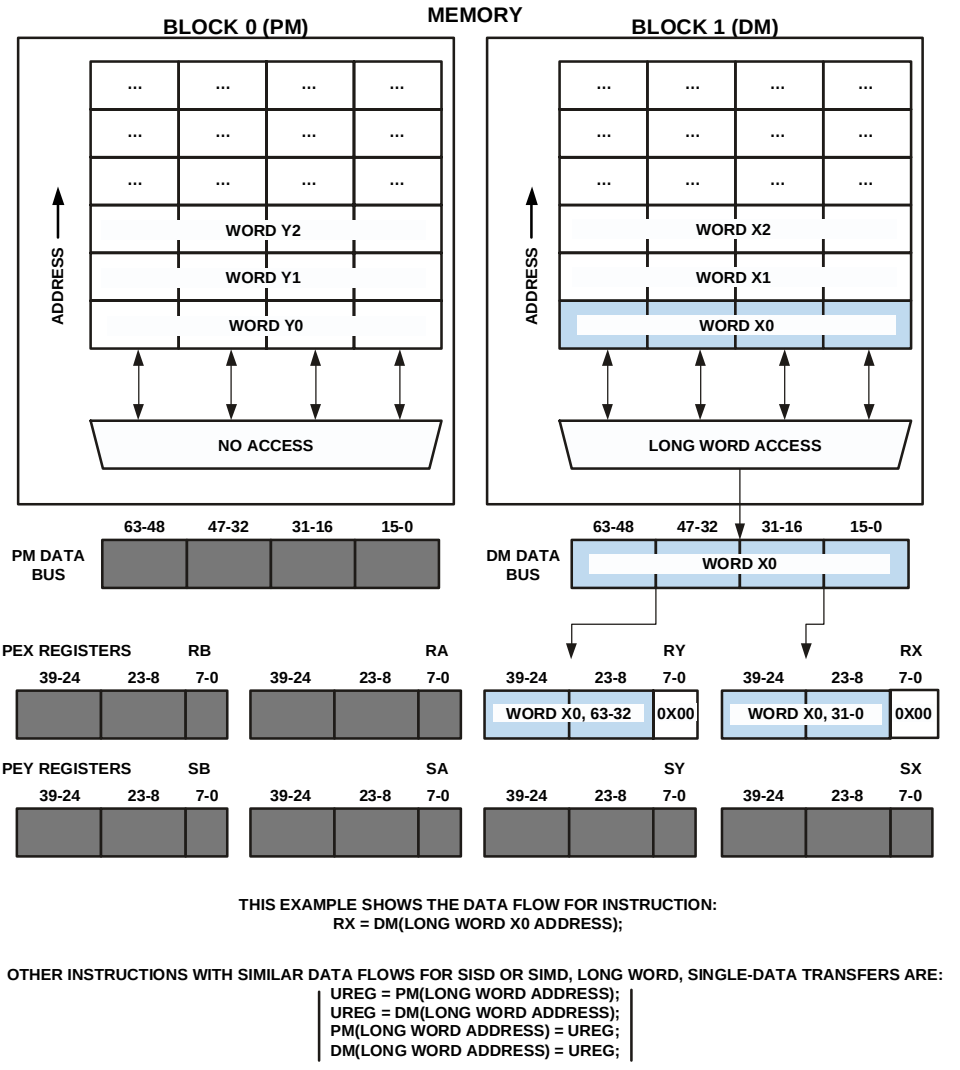


Figure 5-22. Long Word Addressing of Single-Data

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 DSP 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 example targets `PEy` registers when using the syntax `SX` and `SA`. For more information on how neighbor registers (listed in Table 5-3) work, see "Long Word (64-Bit) Accesses" on page 5-28.

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 DSP 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.

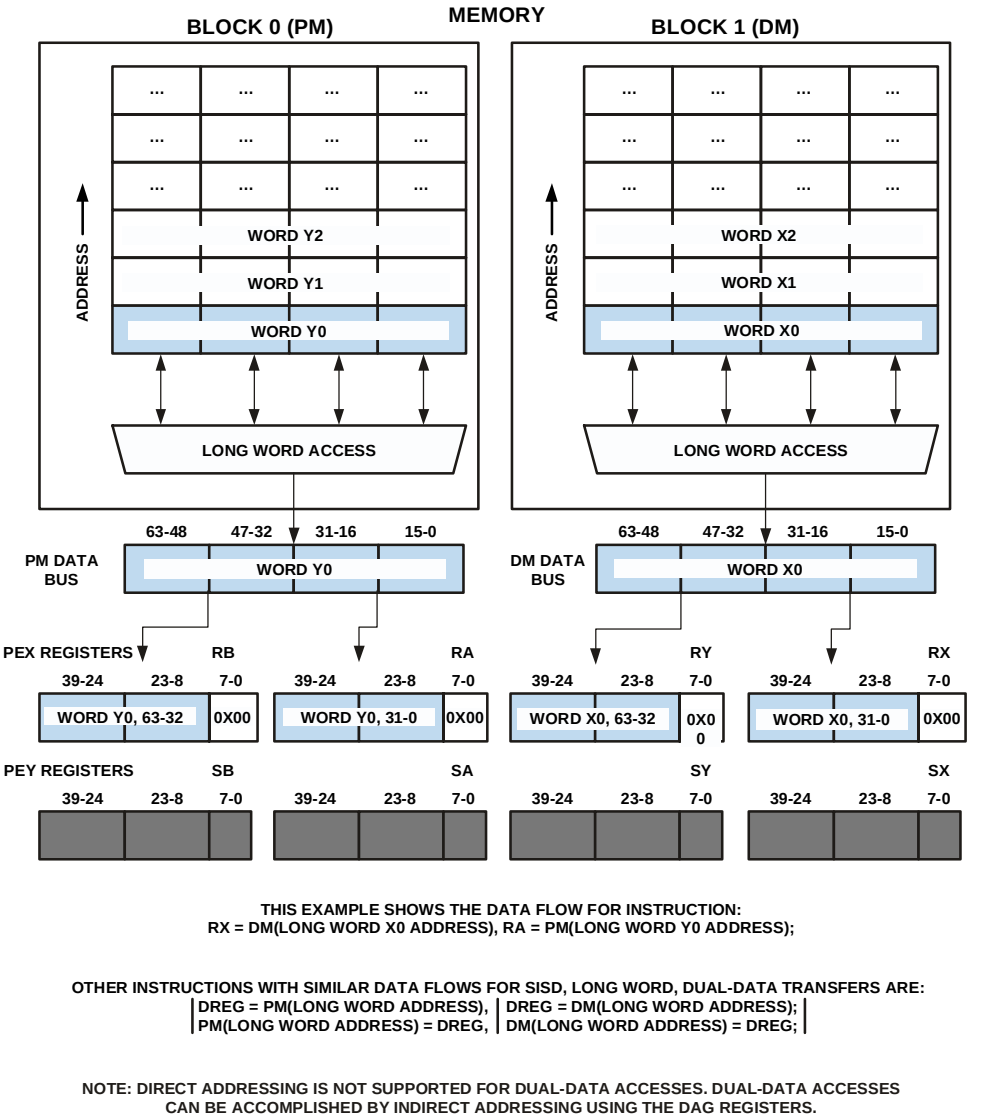


Figure 5-23. Long Word Addressing of Dual-Data in SISD Mode

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 DSP 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`. For more information on how neighbor registers (listed in Table 5-3) work, see "Long Word (64-Bit) Accesses" on page 5-28.

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 5-64.

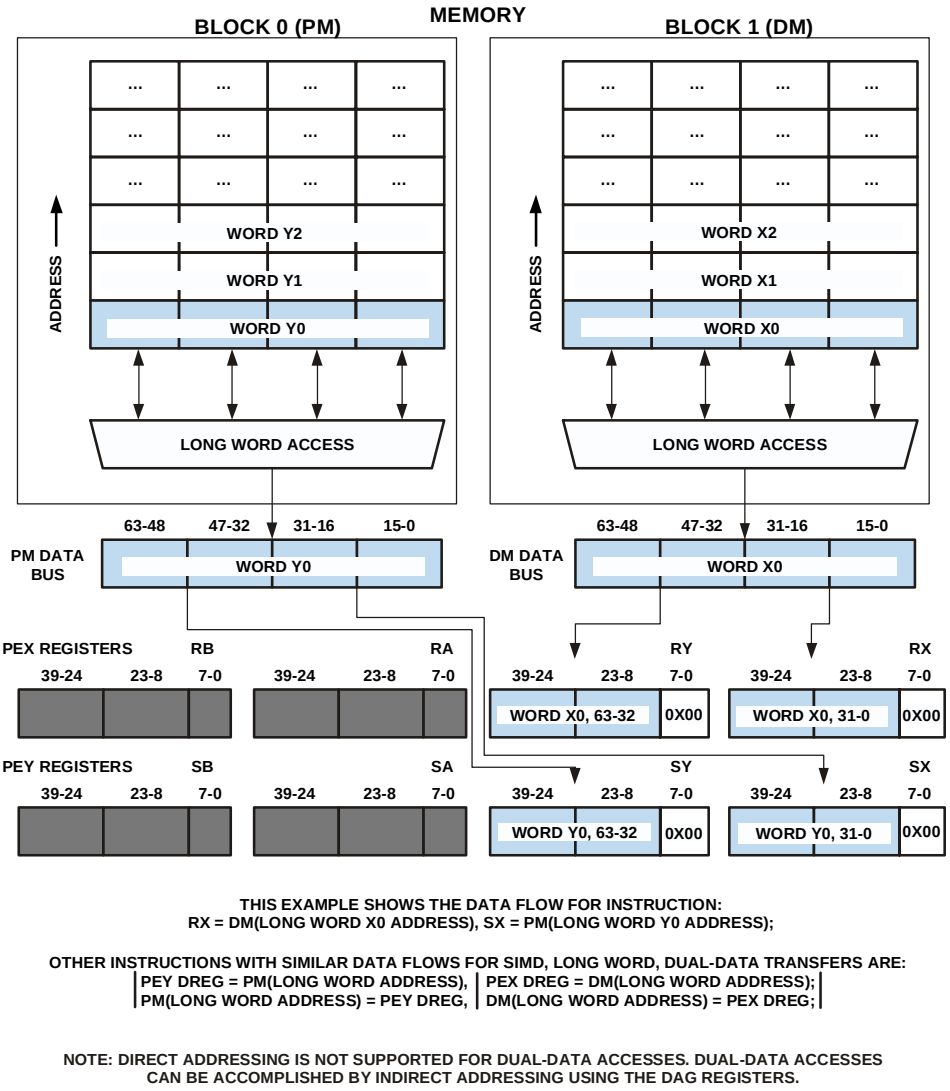


Figure 5-24. Long Word Addressing of Dual-Data in SIMD Mode

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 DSP 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 DSP only performs the access with higher priority. For more information on how the DSP prioritizes accesses, see "Data Register File" on page 2-37.

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 DSP 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.

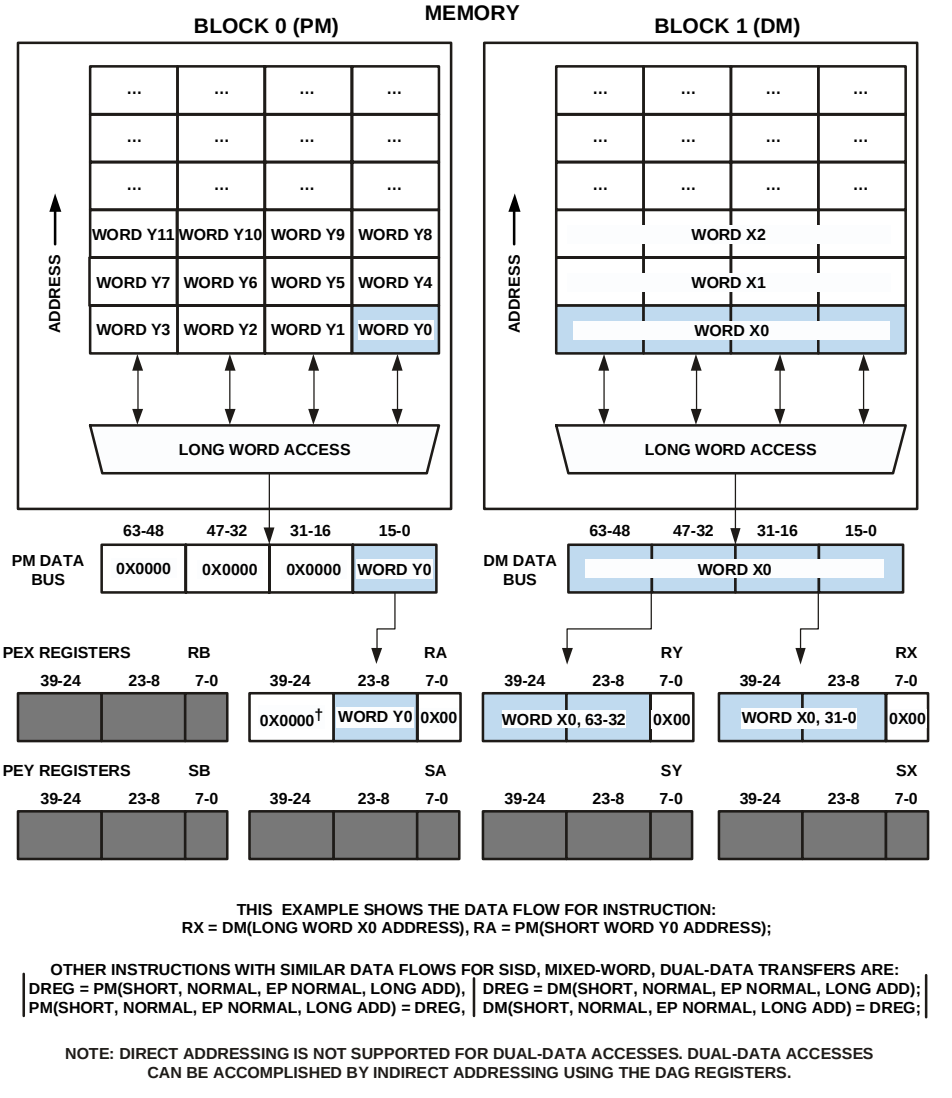
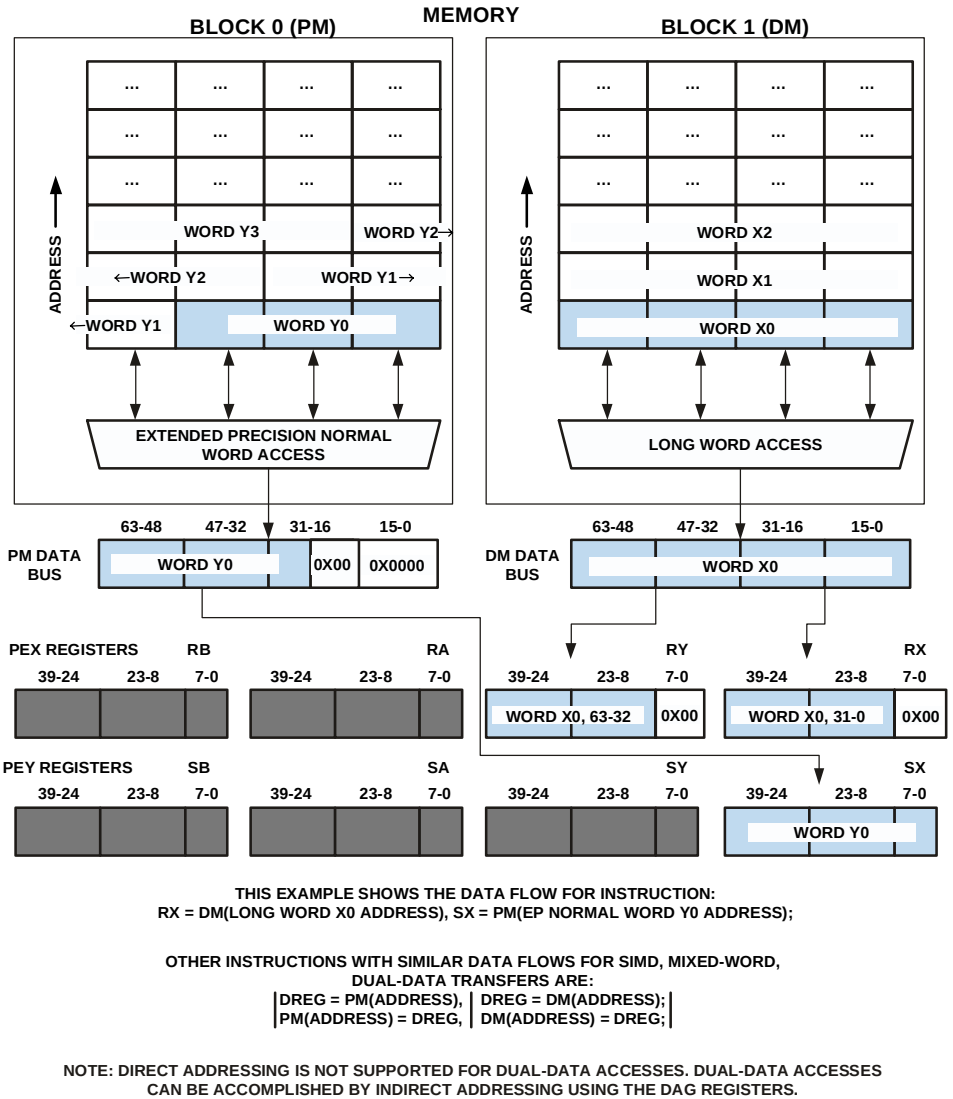


Figure 5-25. Mixed-Word Width Addressing of Dual-Data in SISD Mode



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.

Shadow Write FIFO

Because the DSP'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 Considerations in SIMD Mode

The shadow write FIFO is located between the internal memory array of the ADSP-2126x 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 0x80001 will result in an implicit access to

the next sequential even address value. As shown in Table 5-6, a SIMD mode explicit access to normal word address 0x80001 results in an implicit access to normal word address 0x80002.

Table 5-6. 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

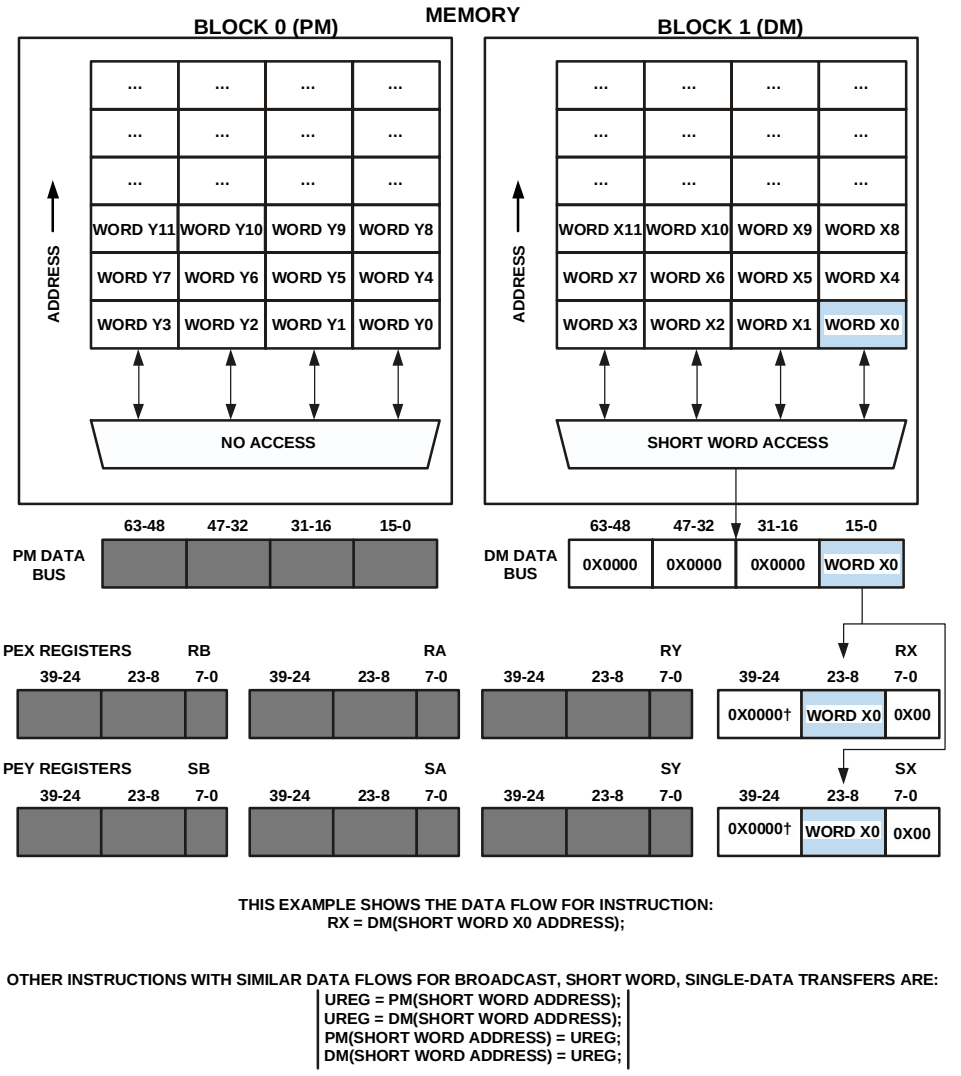


Figure 5-27. Short Word Addressing of Single-Data in Broadcast Load

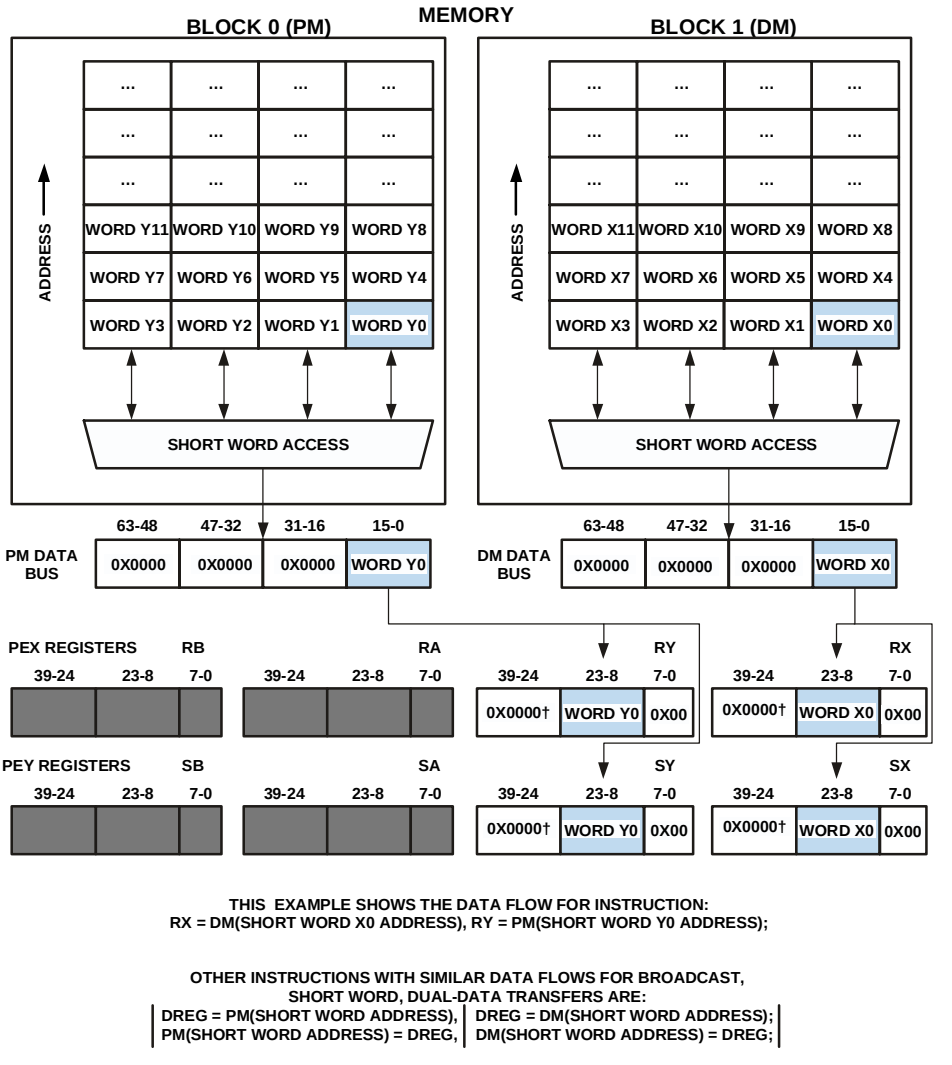


Figure 5-28. Short Word Addressing of Dual-Data in Broadcast Load

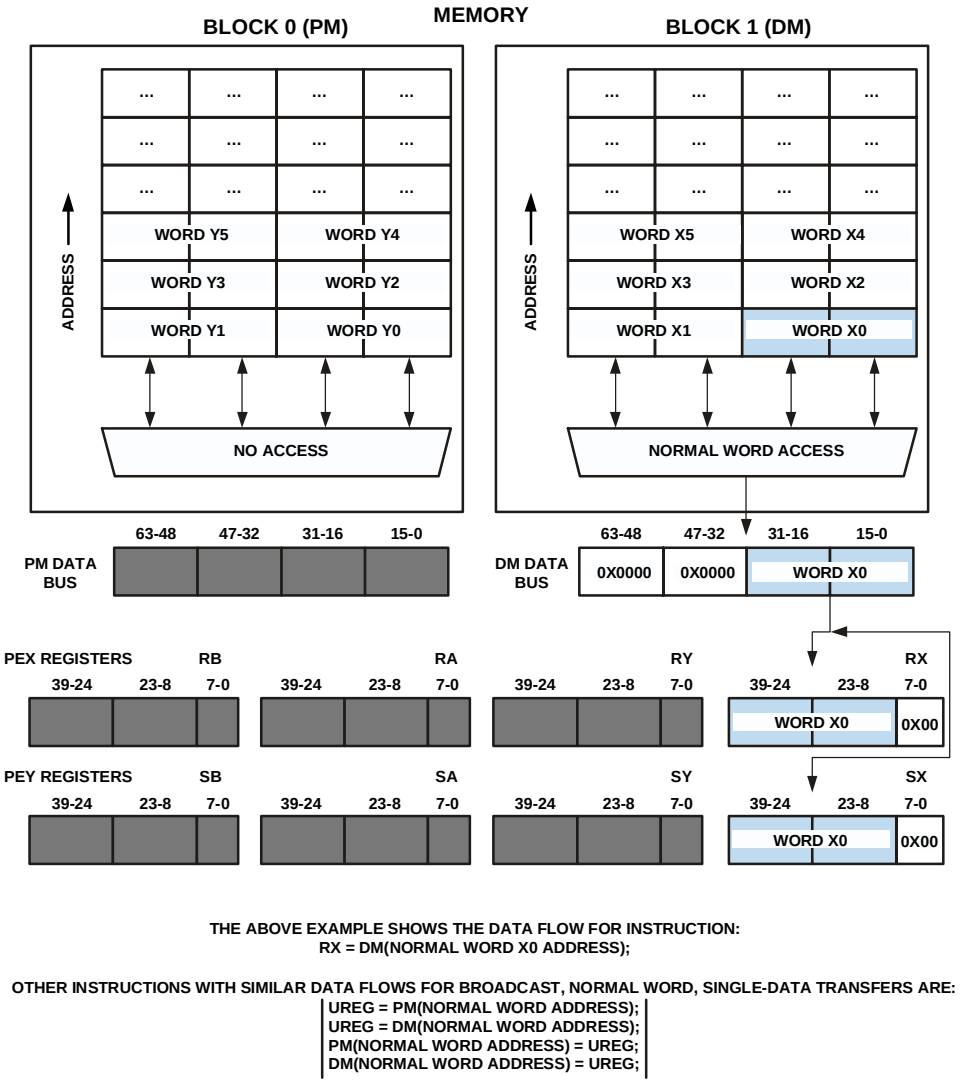


Figure 5-29. Normal Word Addressing of Single-Data in Broadcast Load

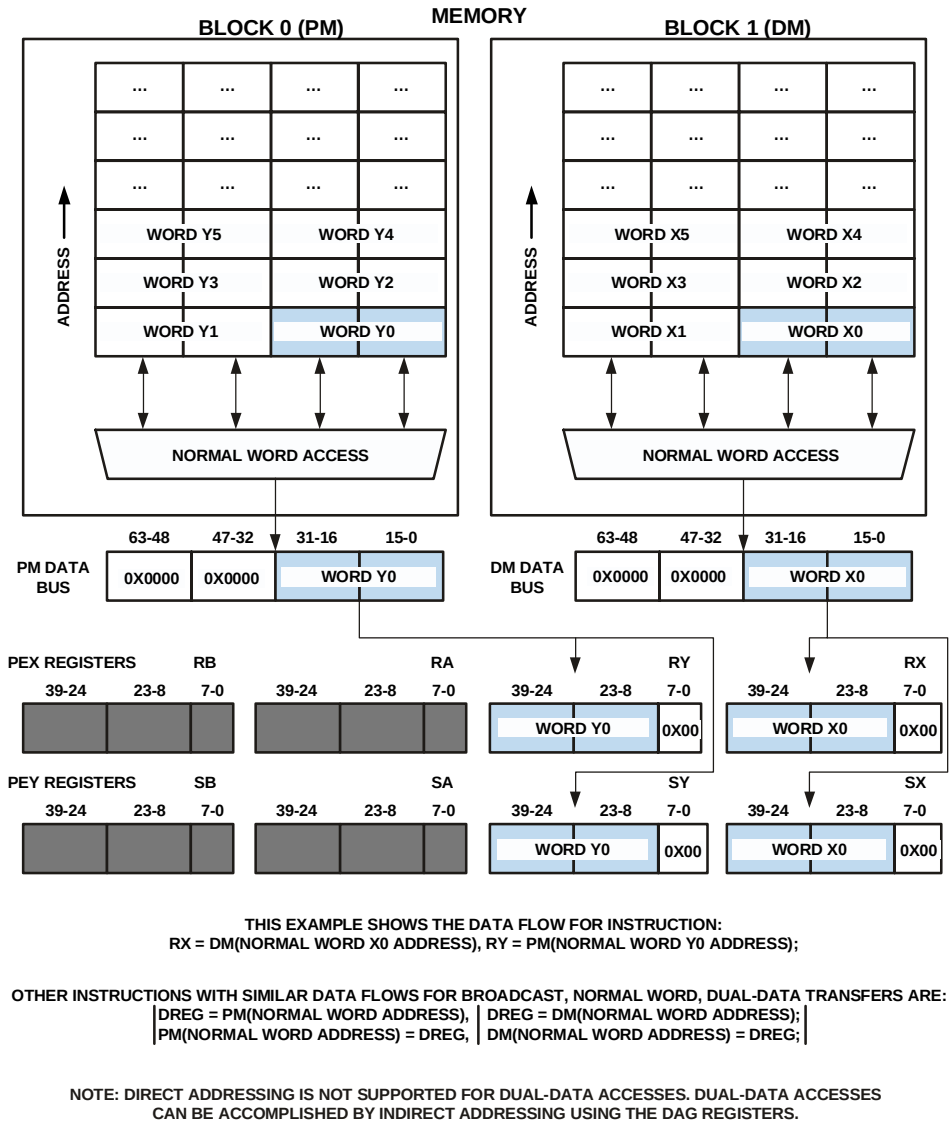


Figure 5-30. Normal Word Addressing of Dual-Data in Broadcast Load

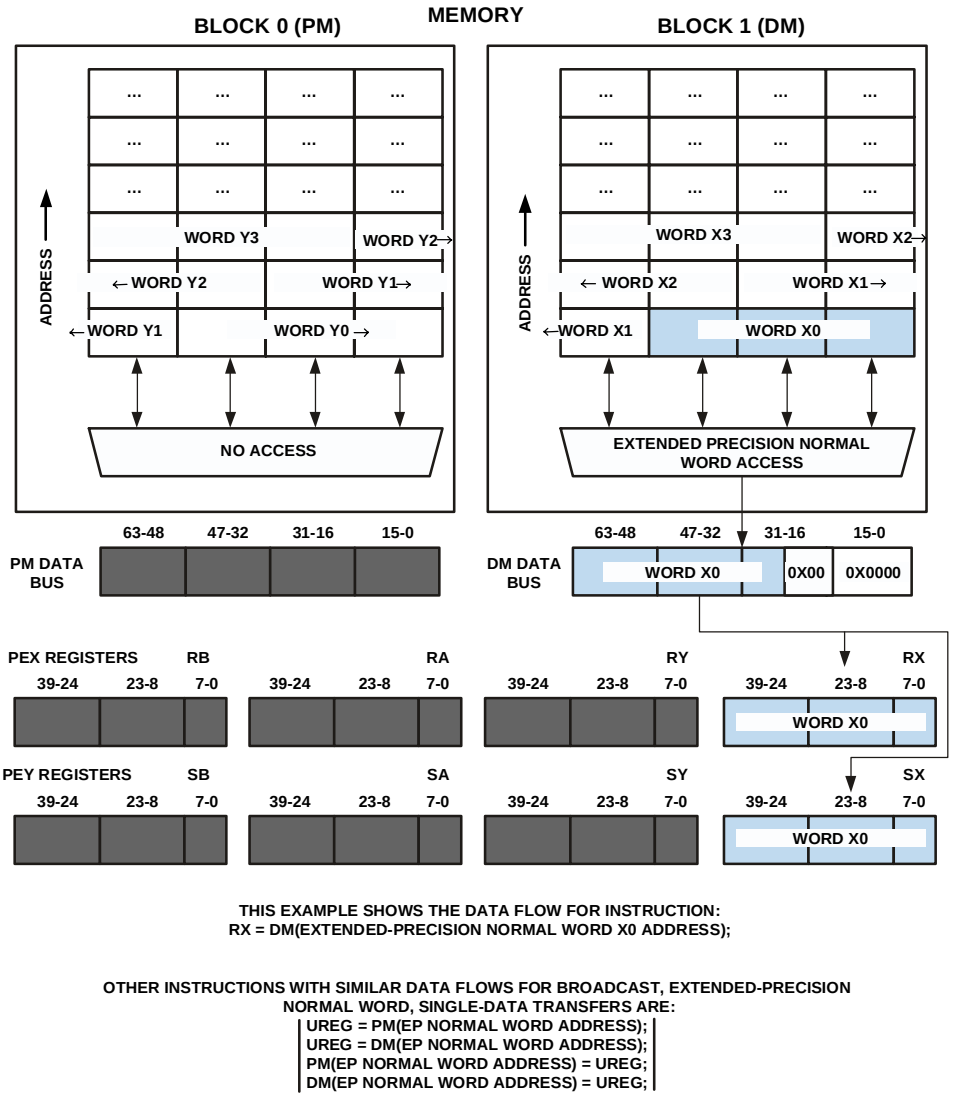
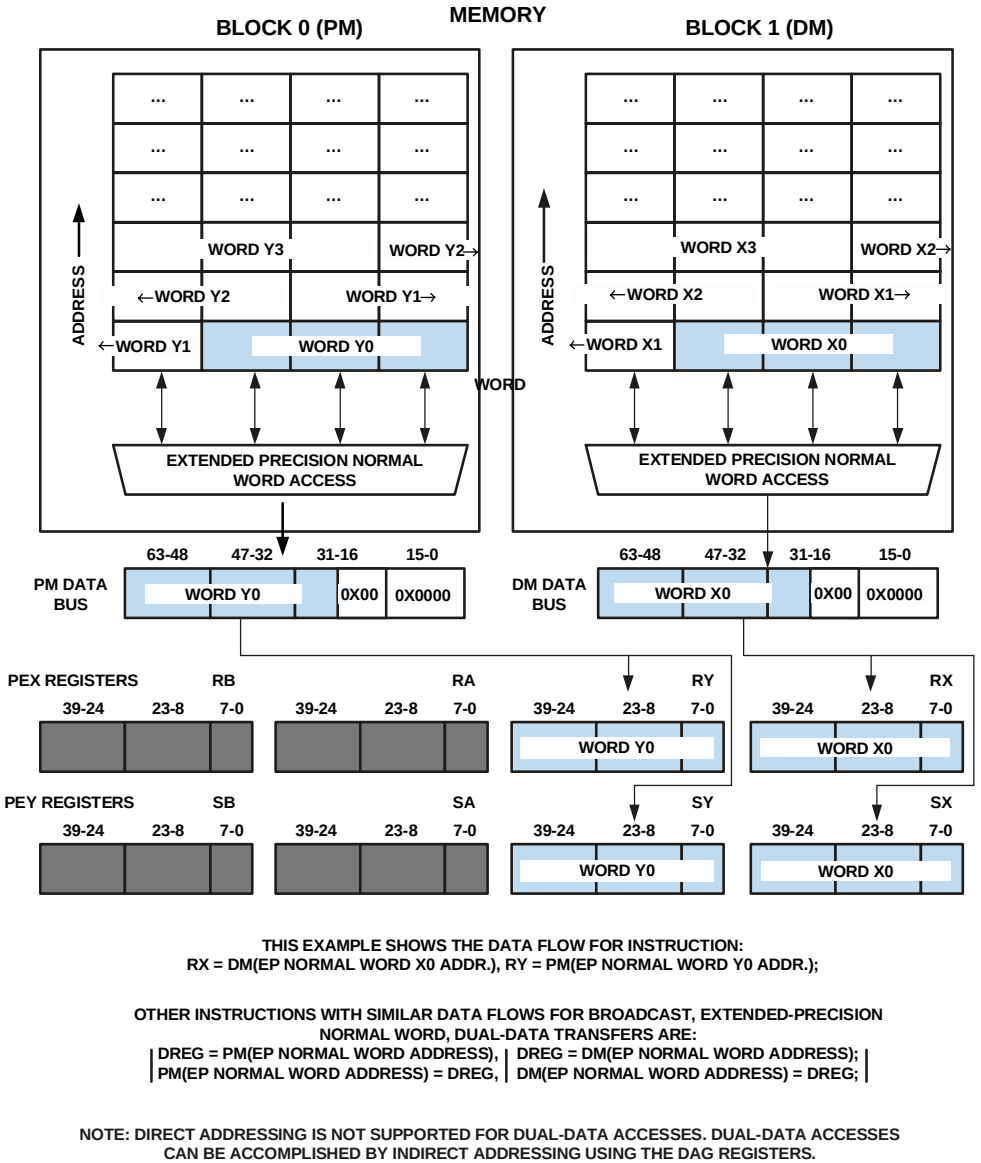


Figure 5-31. Extended-Precision Normal Word Addressing of Single-Data in Broadcast Load



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-PRECISION NORMAL WORD, DUAL-DATA TRANSFERS ARE:

DREG = PM(EP NORMAL WORD ADDRESS), PM(EP NORMAL WORD ADDRESS) = DREG,	DREG = DM(EP NORMAL WORD ADDRESS); 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

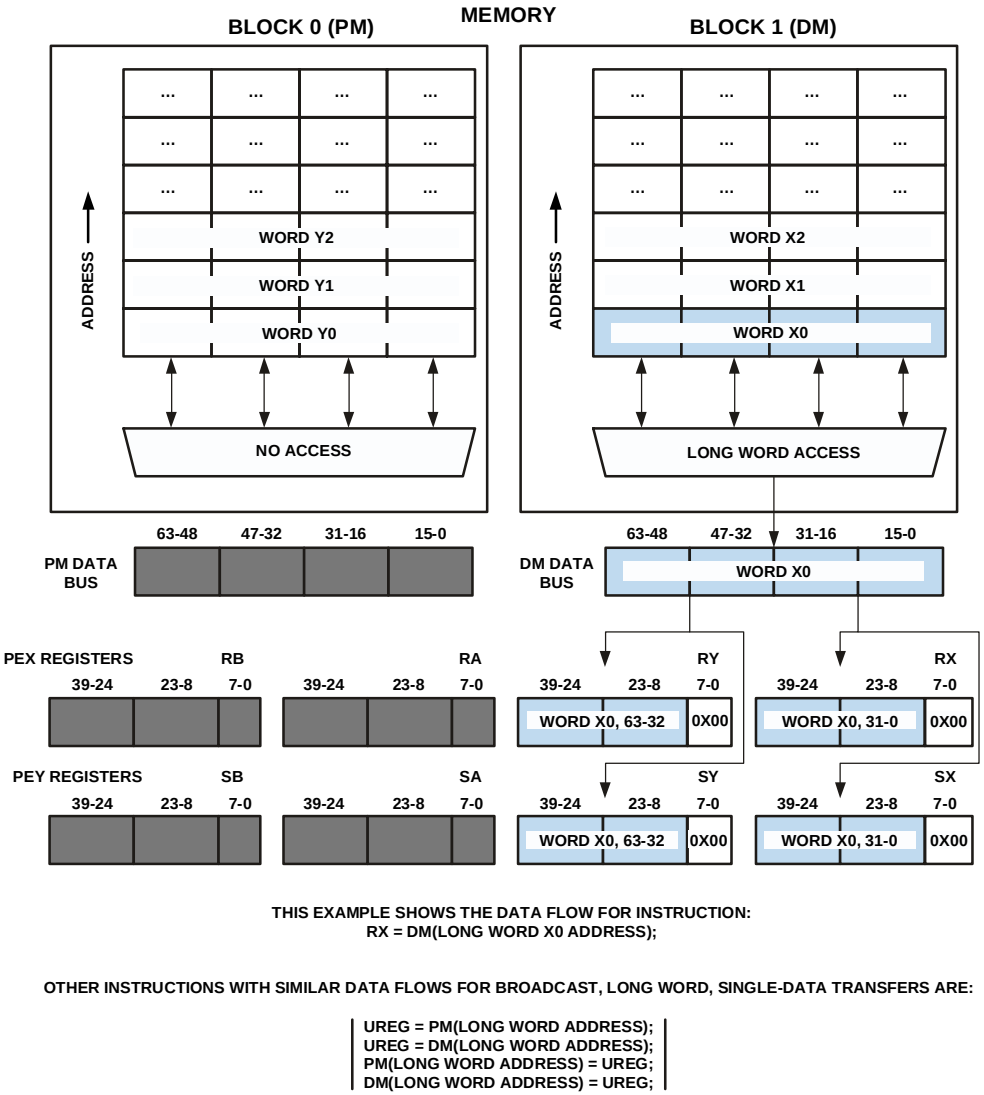


Figure 5-33. Long Word Addressing of Single-Data in Broadcast Load

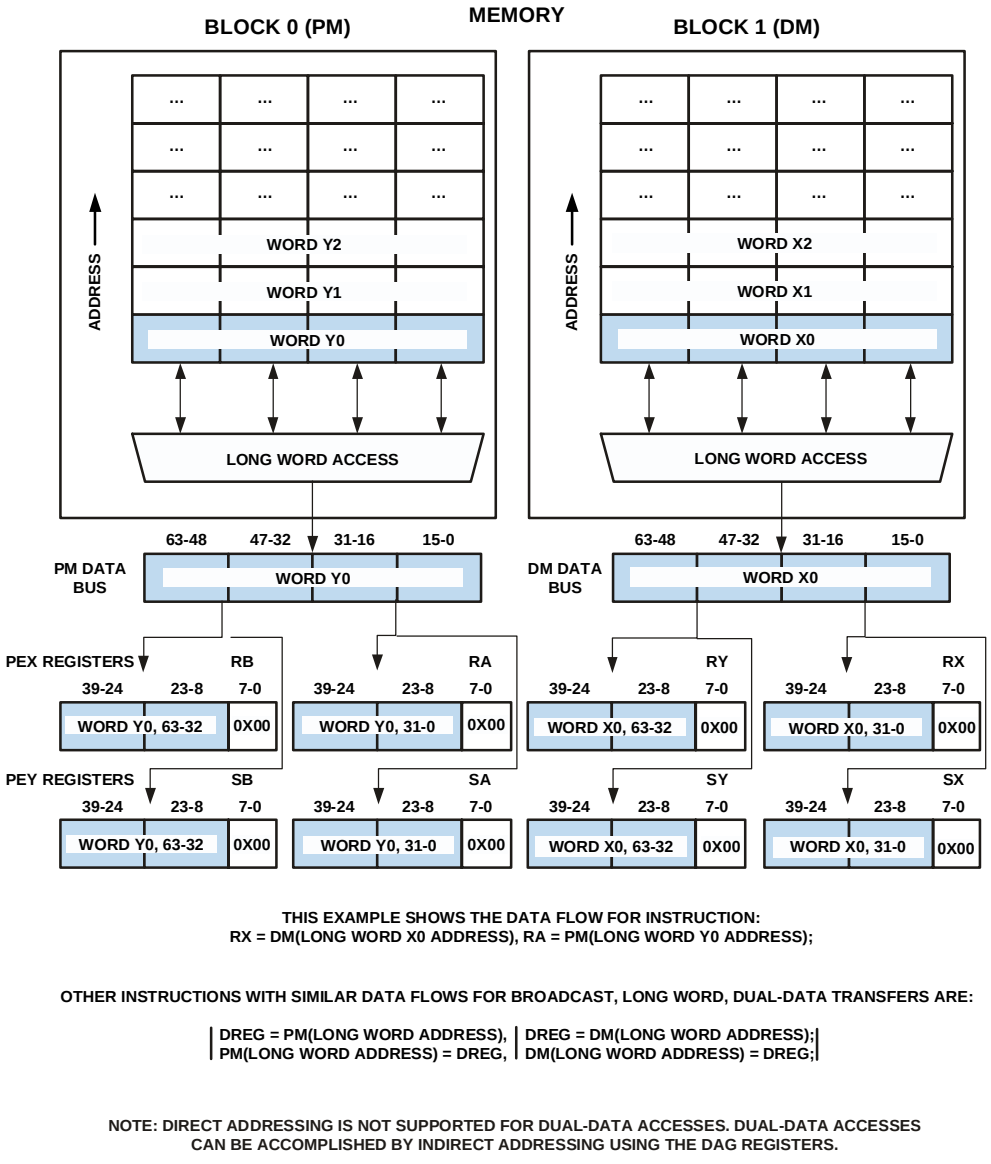


Figure 5-34. Long Word Addressing of Dual-Data in Broadcast Load

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 support, breakpoints, and a register for viewing the revision ID.

JTAG Test Access Port

The emulator uses JTAG boundary scan logic for ADSP-2126x 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 JTAG ICE 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[1]
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
TRST	I	Test Logic Reset: resets the TAP state machine

1 Asynchronous with CLKIN

A Boundary Scan Description language (BSDL) file for the ADSP-2126x 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-2126x 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-2126x processor are described here. For more information, see the IEEE 1149.1 specification and the other documents listed in "References" on page 6-25.

The boundary scan allows a variety of functions to be performed on each input and output signal of the ADSP-2126x 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-2126x processor system clock (`CLKIN`).

The ADSP-2126x 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-2126x processor emulation features are a superset of the ADSP-21160 DSP emulation features. All emulation features supported by previous SHARC DSPs are supported on the ADSP-2126x processor. The set of features on which JTAG ICE designs rely are supported in an identical fashion on ADSP-2126x processor. The DSP can be used with the ADSP-2106x SHARC JTAG ICE hardware.

There are several changes/extensions to the base functionality of the ADSP-2116x DSP emulation capability, which require changes in the JTAG ICE software for ADSP-2126x 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.
- In addition to the IEEE boundary scan functionality, the DSP offers 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-2126x processor. The EMUN, EMUCLK, and EMUCLK2 registers occupy the same *Ureg* address space as the ADSP-2106x DSP. These facilities are read-only by the ADSP-2126x 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 DSP or get updates from the DSP 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 DSP 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 DSP accesses any of the addresses in the address range defined in the Breakpoint registers.

For more information, see “Breakpoint (PSx, DMx, IOx, and EPx) Registers” on page 6-8.

Cycle Count Functionality (EMUCLK) Register

When the emulator is connected to the DSP 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 the 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.

Silicon Revision ID

The ADSP-2126x 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 page 6-6
- The EEMUSTAT register, described on page 6-8
- Breakpoint registers, described on page 6-8
- EEMUIN register, described on page 6-14
- EEMUOUT register, described on page 6-16
- The EMUCLK and EMUCLK2 registers, described on page 6-17

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 10000 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 DSP as defined in the 1149.1 specification. The optional instructions RUNBIST, IDCODE, and USERCODE are not supported by the 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 in this case, 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.

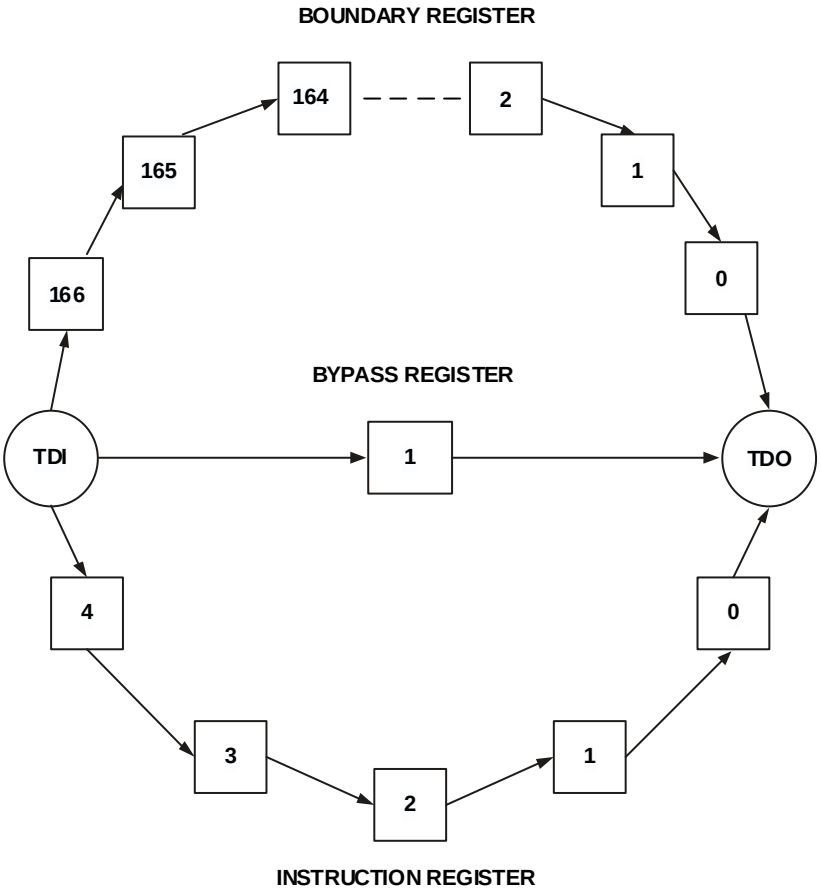


Figure 6-1. Serial Scan Path

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.

Enhanced Emulation Status (EEMUSTAT) Register

The EEMUSTAT register acts as the breakpoint Status register for the ADSP-2126x 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 will be cleared.

Table 6-5 on page 6-15 lists the EEMUSTAT register bits.

Breakpoint Control (BRKCTL) Register

The BRKCTL register controls how breakpoints are used (if the UMODE bit is set). This user-accessible register in the BRKCTL 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 Figure 5-1 on page 5-6 and Table 5-1 on page 5-2. The bits related to the breakpoint register are the same as in the EMUCTL register.

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-2126x processor emulation functionality. Table 6-3 lists the EMUCTL register's bits and describes their function.

Breakpoint (PSx, DMx, IOx, and EPx) Registers

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 DSP is in emulation space or test mode. The Break-

Table 6-3. Emulation Control Register (EMUCTL) Definitions

Bit #	Name	Function
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 recognize external emulator interrupts. (0 = disable, 1 = enable)
2	BKSTOP	**Enable Autostop on Breakpoint.** Enables the ADSP-2126x DSP 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 of the ADSP-2126x processor.** Resets the ADSP-2126x DSP in the same manner as the external $\overline{\text{RESET}}$ pin. 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 DSP 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) Definitions (Cont'd)

Bit #	Name	Function
7	EPSTOP	**Stop I/O Processor EP operation in emulation space.** Disables all EP requests when the DSP 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	**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.

Table 6-3. Emulation Control Register (EMUCTL) Definitions (Cont'd)

Bit #	Name	Function
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	ENBIO	**Enable I/O address breakpoint** see ENBPA bit 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	

Table 6-3. Emulation Control Register (EMUCTL) Definitions (Cont'd)

Bit #	Name	Function
34	NOBOOT	**No power-up boot on reset.** Forces the DSP into the No boot mode. In this mode, the processor does not boot load, but begins fetching instructions from 0x0080 0004 in external 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 bit.** The BHO control bit overrides the BHD bit in SYSCON, disabling BHD's control over core access of data buffer behavior. Note that the default (reset) state of BHD is now set for the DSP, a change from ADSP-2106x. (0 = normal BHD operation, 1 = override BHD operation)
37	MTST	**Memory Test Enable Bit.** Enables scanning of data for to the latches used for memory test. (0 = normal operation, 1 = enable memory test)
38, 39	Reserved	

point registers vary in size according to the address type: instruction (24-bit address), data (32-bit address), or I/O data (19-bit address)—Table 6-7 on page 6-22 shows the sizes.

The ADSP-2126x 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 eight 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 6-7.

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 on page 6-22. 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 one monitors 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 6-7 on page 6-22.

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
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.

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 DSP without interrupt-

Table 6-5. EEMUSTAT (Breakpoint Status) Register Definitions

Bits	Name	Function
0	STATPA	**Program memory Data Breakpoint Hit**[1] 1 = Program memory breakpoint occurs 0 = No program memory breakpoint occurs
1	STATDA0	**Data Memory Breakpoint Hit**[1] 1 = Data memory #0 breakpoint occurs 0 = No data memory #0 breakpoint occurs
2	STATDA1	**Data Memory Breakpoint Hit**[1] 1 = Data memory #1 breakpoint occurs 0 = No Data memory #1 breakpoint occurs
3	STATIA0	**Instruction Address Breakpoint Hit**[1] 1 = Instruction address #0 breakpoint occurs 0 = no Instruction address #0 breakpoint occurs
4	STATIA1	**Instruction Address Breakpoint Hit**[1] 1 = Instruction address #1 breakpoint occurs 0 = no Instruction address #1 breakpoint occurs
5	STATIA2	**Instruction Address Breakpoint Hit**[1] 1 = Instruction address #2 breakpoint occurs 0 = no Instruction address #2 breakpoint occurs
6	STATIA3	**Instruction Address Breakpoint Hit**[1] 1 = Instruction address #3 breakpoint occurs 0 = no Instruction address #3 breakpoint occurs
7	STATIO	**I/O Address Breakpoint Hit** 1 = I/OX address breakpoint occurs 0 = no I/OX address breakpoint occurs
8	Reserved**1**	
9	EEMU-OUTIRQEN	**Enhanced Emulation EEMUOUT Interrupt Enable**[2] 1 = EEMUOUT interrupt enable 0 = EEMUOUT interrupt disable Note: Interrupts are of low priority interrupts

Table 6-5. EEMUSTAT (Breakpoint Status) Register Definitions (Cont'd)

Bits	Name	Function
10	EEMUOUTRDY	**Enhanced Emulation EEMUOUT Ready**[3] 1 = EEMUOUT FIFO contains valid data 0 = EEMUOUT FIFO is empty
11	EEMUOUTFULL	**Enhanced Emulation EEMUOUT FIFO Status**[3] 1 = EEMUOUT FIFO FULL 0 = EEMUOUT FIFO is not FULL
12	EEMUINFULL	**Enhanced Emulation EEMUIN Register Status**[4] 1 = EEMUIN register full 0 = EEMUIN register is empty
13	EEMUENS	**Enhanced Emulation Feature Enable**[4] 1 = Enhanced emulation feature enable 0 = Enhanced emulation feature disable
14	OSPIDENS	**OSPID Register Enable**[4] 1 = OSPID register enable 0 = OSPID register disable
15	EEMUINENS	**EEMUIN Interrupt Enable.** 1 = EEMUIN interrupt enable 0 = EEMUIN interrupt disable
31:16	Reserved for future use.	

1 Internal hardware sets this bit.
2 This bit is set and reset by the core.
3 The FIFO controller sets and resets this bit.
4 Internal hardware sets and resets this bit.

ing 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

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 DSP is in emulation space. These registers are read-only from normal-space and can be written only when the ADSP-2126x 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 DSP 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.

Boundary Register

The Boundary register is 163 bits long. This section defines the latch type and function of each position in the scan path. The positions are numbered with 0 being the first bit output (closest to TDO) and 162 being the last (closest to TDI). When working with boundry scan registers keep the following points in mind:

Scan position 0 (CLK_CFG0); this end is closest to TDO (scan in first).

Scan position 162 (SPARE); this end is closest to TDI (scan in last).

Output enables:

1 = Drive the associated signals during the EXTEST and INTEST instructions.

0 = Three-state the associated signals during the EXTEST and INTEST instructions

The CLKIN signal can be sampled but not controlled (read-only). CLKIN continues to clock the ADSP-2126x processor no matter which instruction is enabled.

Table 6-6. JTAG Boundary Register

Scan #	Signal Name	Latch Type	Scan #	Signal Name	Latch Type
0	NC(I)	OUTP (Closest to TDO, scan in first)	84	SDATA0B(I/O)	IN
1	CLK_CFG0(I)	OUTP	85	SCLK0(I/O)	OUTP
2	CLK_CFG0(I)	OE	86	SCLK0(I/O)	OE
3	CLK_CFG0(I)	IN	87	SCLK0(I/O)	IN
4	CLK_CFG1(I)	OUTP	88	SFS0(I/O)	OUTP
5	CLK_CFG1(I)	OE	89	SFS0(I/O)	OE
6	CLK_CFG1(I)	IN	90	SFS0(I/O)	IN
7	BOOTCFG[0](I)	OUTP	91	SDATA1A(I/O)	OUTP
8	BOOTCFG[0](I)	OE	92	SDATA1A(I/O)	OE
9	BOOTCFG[0](I)	IN	93	SDATA1A(I/O)	IN
10	BOOTCFG[1](I)	OUTP	94	SDATA1B(I/O)	OUTP
11	BOOTCFG[1](I)	OE	95	SDATA1B(I/O)	OE
12	BOOTCFG[1](I)	IN	96	SDATA1B(I/O)	IN
13	FLG0(I/O)	OUTP	97	SCLK1(I/O)	OUTP
14	FLG0(I/O)	OE	98	SCLK1(I/O)	OE
15	FLG0(I/O)	IN	99	SCLK1(I/O)	IN
16	FLG1(I/O)	OUTP	100	SFS1(I/O)	OUTP
17	FLG1(I/O)	OE	101	SFS1(I/O)	OE
18	FLG1(I/O)	IN	102	SFS1(I/O)	IN
19	DATA[7](I/O)	OUTP	103	SDATA2A(I/O)	OUTP

Table 6-6. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type	Scan #	Signal Name	Latch Type
20	DATA[7](I/O)	OE	104	SDATA2A(I/O)	OE
21	DATA[7](I/O)	IN	105	SDATA2A(I/O)	IN
22	DATA[6](I/O)	OUTP	106	SDATA2B(I/O)	OUTP
23	DATA[6](I/O)	OE	107	SDATA2B(I/O)	OE
24	DATA[6](I/O)	IN	108	SDATA2B(I/O)	IN
25	DATA[5](I/O)	OUTP	109	SDATA2C(I/O)	OUTP
26	DATA[5](I/O)	OE	110	SDATA2C(I/O)	OE
27	DATA[5](I/O)	IN	111	SDATA2C(I/O)	IN
28	DATA[4](I/O)	OUTP	112	SDATA2D(I/O)	OUTP
29	DATA[4](I/O)	OE	113	SDATA2D(I/O)	OE
30	DATA[4](I/O)	IN	114	SDATA2D(I/O)	IN
31	DATA[3](I/O)	OUTP	115	SCLK2(I/O)	OUTP
32	DATA[3](I/O)	OE	116	SCLK2(I/O)	OE
33	DATA[3](I/O)	IN	117	SCLK2(I/O)	IN
34	DATA[2](I/O)	OUTP	118	SFS2(I/O)	OUTP
35	DATA[2](I/O)	OE	119	SFS2(I/O)	OE
36	DATA[2](I/O)	IN	120	SFS2(I/O)	IN
37	DATA[1](I/O)	OUTP	121	SDATA3A(I/O)	OUTP
38	DATA[1](I/O)	OE	122	SDATA3A(I/O)	OE
39	DATA[1](I/O)	IN	123	SDATA3A(I/O)	IN
40	DATA[0](I/O)	OUTP	124	SDATA3B(I/O)	OUTP
41	DATA[0](I/O)	OE	125	SDATA3B(I/O)	OE
42	DATA[0](I/O)	IN	126	SDATA3B(I/O)	IN
43	WR_B(I/O)	OUTP	127	SDATA3C(I/O)	OUTP

Table 6-6. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type	Scan #	Signal Name	Latch Type
44	WR_B(I/O)	OE	128	SDATA3C(I/O)	OE
45	WR_B(I/O)	IN	129	SDATA3C(I/O)	IN
46	RD_B(I/O)	OUTP	130	SDATA3D(I/O)	OUTP
47	RD_B(I/O)	OE	131	SDATA3D(I/O)	OE
48	RD_B(I/O)	IN	132	SDATA3D(I/O)	IN
49	ALE(I/O)	OUTP	133	SCLK3(I/O)	OUTP
50	ALE(I/O)	OE	134	SCLK3(I/O)	OE
51	ALE(I/O)	IN	135	SCLK3(I/O)	IN
52	ADDR[7](I/O)	OUTP	136	SFS3(I/O)	OUTP
53	ADDR[7](I/O)	OE	137	SFS3(I/O)	OE
54	ADDR[7](I/O)	IN	138	SFS3(I/O)	IN
55	ADDR[7](I/O)	OUTP	139	FLG2(I/O)	OUTP
56	ADDR[7](I/O)	OE	140	FLG2(I/O)	OE
57	ADDR[7](I/O)	IN	141	FLG2(I/O)	IN
58	ADDR[6](I/O)	OUTP	142	FLG3(I/O)	OUTP
59	ADDR[6](I/O)	OE	143	FLG3(I/O)	OE
60	ADDR[6](I/O)	IN	144	FLG3(I/O)	IN
61	ADDR[5](I/O)	OUTP	145	RESET_B(I)	OUTP
62	ADDR[5](I/O)	OE	146	RESET_B(I)	OE
63	ADDR[5](I/O)	IN	147	RESET_B(I)	IN
64	ADDR[4](I/O)	OUTP	148	SPIDS(I)	OUTP
65	ADDR[4](I/O)	OE	149	SPIDS(I)	OE
66	ADDR[4](I/O)	IN	150	SPIDS(I)	IN
67	ADDR[3](I/O)	OUTP	151	SPICK(I/O)	OUTP

Table 6-6. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type	Scan #	Signal Name	Latch Type
68	ADDR[3](I/O)	OE	152	SPICK(I/O)	OE
69	ADDR[3](I/O)	IN	153	SPICK(I/O)	IN
70	ADDR[2](I/O)	OUTP	154	MISO(I/O)	OUTP
71	ADDR[2](I/O)	OE	155	MISO(I/O)	OE
72	ADDR[2](I/O)	IN	156	MISO(I/O)	IN
73	ADDR[1](I/O)	OUTP	157	MOSI(I/O)	OUTP
74	ADDR[1](I/O)	OE	158	MOSI(I/O)	OE
75	ADDR[1](I/O)	IN	159	MOSI(I/O)	IN
76	ADDR[0](I/O)	OUTP	160	CLKOUT(O)	OUTP
77	ADDR[0](I/O)	OE	161	CLKOUT(O)	OE
78	ADDR[0](I/O)	IN	162	CLKOUT(O)	IN
79	SDATA0A(I/O)	OUTP	163	EMU_B(O)	OUTP
80	SDATA0A(I/O)	OE	164	EMU_B(O)	OE
81	SDATA0A(I/O)	IN	165	EMU_B(O)	IN
82	SDATA0B(I/O)	OUTP	166	SPARE	Closest to TDI scan in last
83	SDATA0B(I/O)	OE			

Built-In Self-Test Operation (BIST)

No self-test functions are supported by the ADSP-2126x processor.

EMUCTL Shift Register

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 DSP's emulation functionality.

Table 6-7 lists the EMUCTL's register's bits and describes their functionality.

Table 6-7. Emulation Control Register (EMUCTL) Definitions

Bits	Name	Function
0	EMUENA	Enable auto-stop on breakpoint
1	EIRQENA	Enable single-step mode
2	BKSTOP	Enable the breakout pin functionality
3	SS	Enable the single-step mode
4	SYSRST	Software reset of ADSP-2126x
5	ENBRKOUT	Enable the breakout pin functionality
6	IOSTOP	Stop I/O processor DMAs in emulation space
7	Reserved	
8	NEGPA1	Negate program memory data address breakpoint
9	NEGDA1	Negate data memory address breakpoint #1
10	NEGDA2	Negate data memory address breakpoint #2
11	NEGIA1	Negate instruction address breakpoint #1
12	NEGIA2	Negate instruction address breakpoint #2
13	NEGIA3	Negate instruction address breakpoint #3
14	Reserved	
15	NEGIO1	Negate I/O address breakpoint
16	Reserved	
17	ENBPA	Enable program memory data address breakpoint
18	ENBDA	Enable data memory address breakpoint
19	ENBIA	Enable instruction address breakpoint
20	Reserved[1]	
21	Reserved	
23:22	PA1MODE	PA1 breakpoint triggering mode

Table 6-7. Emulation Control Register (EMUCTL) Definitions (Cont'd)

Bits	Name	Function
25:24	DA1MODE	DA1 breakpoint triggering mode
27:26	DA2MODE	DA2 breakpoint triggering mode
29:28	IO1MODE	IO1 breakpoint triggering mode
31:30	EP1MODE	EP1 breakpoint triggering mode
32	ANDBKP	AND the composite breakpoints
33	Reserved	
34	NOBOOT	NO power-up boot on reset
35	TMODE	Test mode bit
36	BHO	Buffer hang override
37	MTST	Fuse test
39:38	TE_IODISABLE	00 = IOX and IOY breakpoint disabled 01 = IOY breakpoint enabled IOX breakpoint disabled 10 = IOY breakpoint disabled IOX breakpoint enabled 11 = IOY and IOX breakpoint enabled

1 The ENBIO bit, provided in previous SHARC DSP products, is not supported in the DSP; it has been replaced with TE_IODISABLE (bits 39:38).

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 by the user 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.

EMUIDLE Instruction

The EMUIDLE instruction places the DSP 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.

OSPID Register

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 register includes a control bit, (OSPIDEN). This is 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-2126x processor JTAG ICE emulator uses the TAP and boundary scan as a way to access the processor in the target system. The JTAG ICE 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-2126x DSP 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 7-7
- "Pulse Width Count and Capture Mode (WDTH_CAP)" on page 7-10
- "External Event Watchdog Mode (EXT_CLK)" on page 7-12

Timer Architecture

Each timer has one dedicated bidirectional chip signal, `TIMERx`. The three timer signals are connected to the 20 Digital Audio Interface (DAI) pins through the Signal Routing Unit (SRU). 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 Configuration (`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 one common status and control register, the Timer Global Status and Control (TMSTAT) register.

For information on the Timer registers, see "Timer Registers" on page A-61.

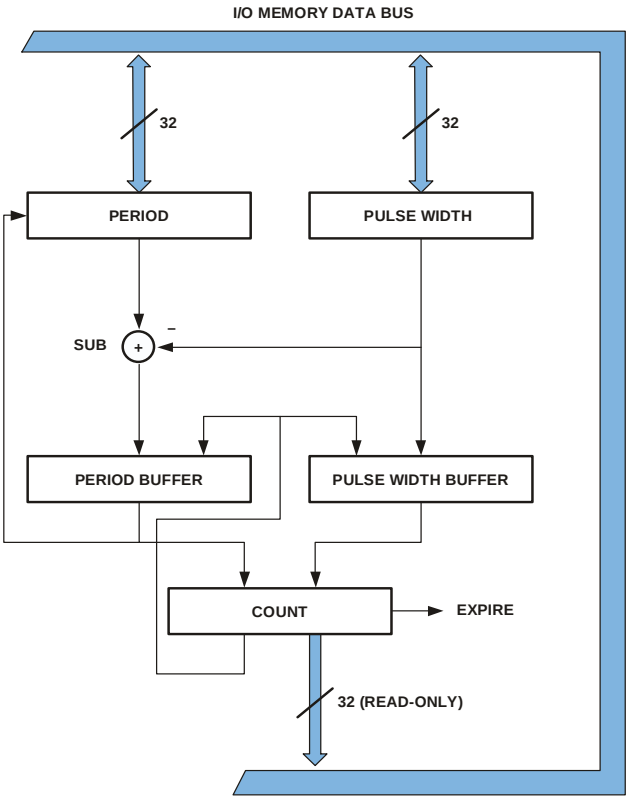


Figure 7-1. Timer Block Diagram

When clocked internally, the clock source is the ADSP-2126x DSP's core clock (CCLK). 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 CCLK = 200 MHz:

maximum period = 2 x (2^{31} – 1) x 5 ns = 20 seconds.

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.

After the timer has been enabled, its TIMxEN bit is set (= 1). The timer then starts counting three core clock cycles 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.

Table 7-1. Timer Global Status and Control (TMSTAT) Register Bits

Bit(s)	Name	Definition
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 will return a zero.

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 A-61.

The timer enable and disable timing appears in Figure 7-2.

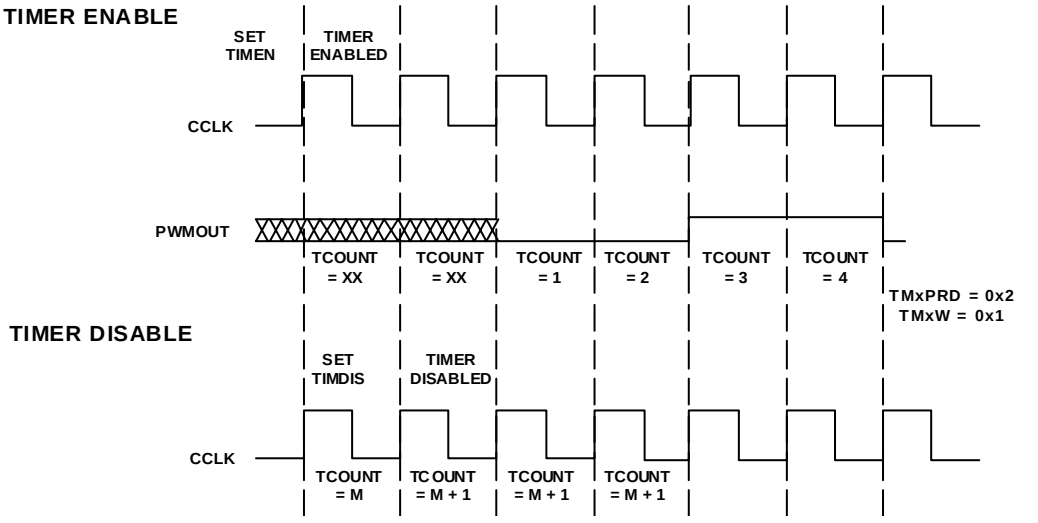


Figure 7-2. Timer PWM Enable and Disable Timing

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 DSP 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 7-3 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

On invalid conditions, the timer sets both the TMxOVF 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 FFF. Instead

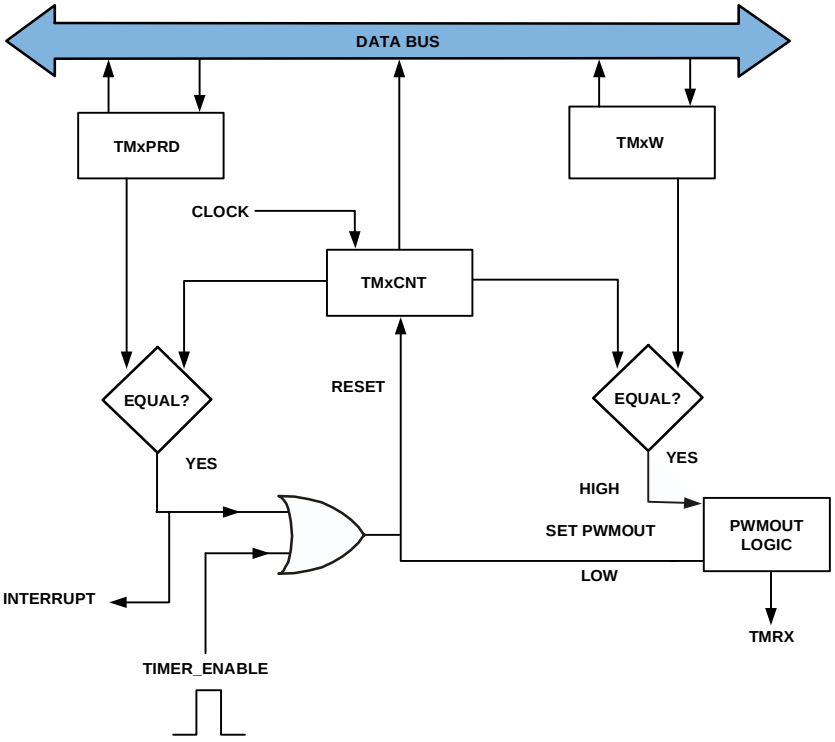


Figure 7-3. Timer Flow Diagram - PWM_OUT Mode

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 DSP 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 two CCLK clock cycles.

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.

Using a General-Purpose Timer as a Core Timer

Programs can use a general-purpose timer as a core timer. When in this mode, the timer can also generate a periodic interrupt in a fashion similar to the core timer. In this case there is no need to route the timer signal to an external pin.

To implement this behavior, it is necessary to set the TIMODEPWM bits, the PRDCNT bit, and the IRQEN bit in the applicable TMxCTL register. The period at which the interrupt is latched is the pulse period (2 x value in TMxPRD register) in core cycles. Even though the TMxW register is not used in this case, it is necessary to initialize it to a nonzero value less than the value in the TMxPRD register for correct operation.

Unlike the core timer, programs must manually clear the interrupt in the TMSTAT register for each interrupt that is serviced.

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/2.

Figure 7-4 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 0001 and does not start counting until it detects the leading edge on the TIMERx signal.

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

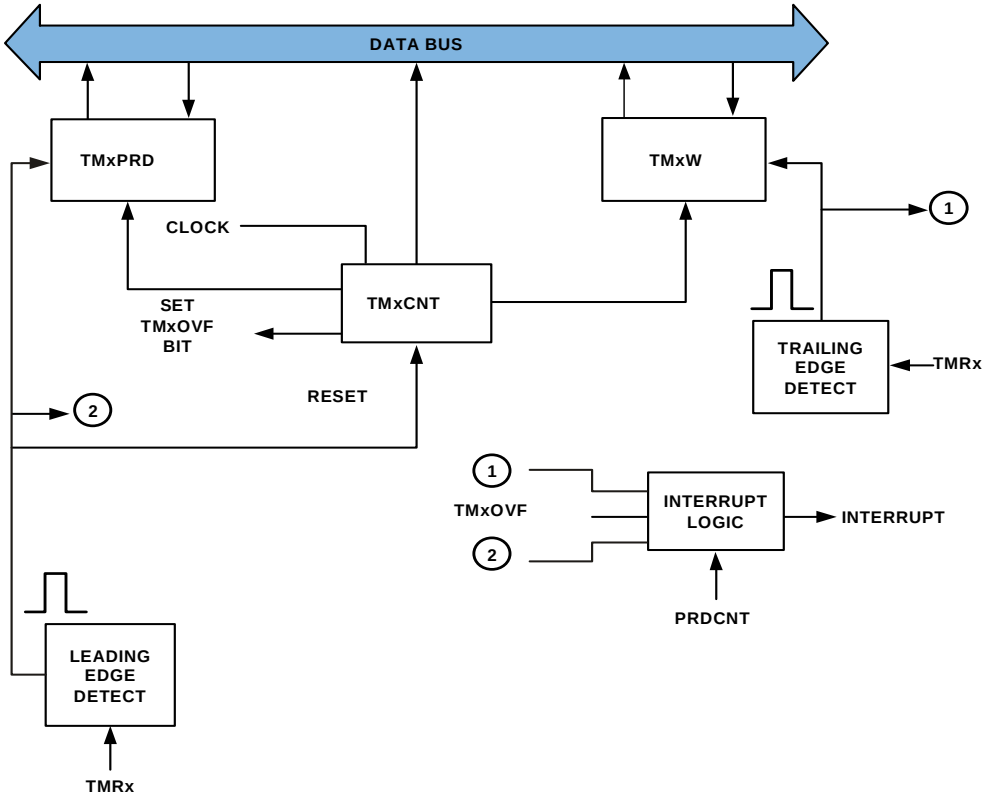


Figure 7-4. Timer Flow Diagram – WDTH_CAP Mode

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 0001 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 TMxOVF 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 DSP 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:

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-2126x DSP. The first listing, Listing 7-1, 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.

The second listing, Listing 7-2, 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.

Listing 7-1. 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 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. External Watchdog 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_0d       0x2C
#define TIMER0_0e       0x14
#define PBEN_HIGH_0f    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_0d;dm(SRU_PIN0) = r0;

/* Enable DAI pin 1 as an output */
r0 = PBEN_HIGH_0f;
```

```
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_0e<<TIMER1_I);
```

```
dm(SRU_EXT_MISCB)=r0;


ustat3 = TIMODEW|       /* PWM Out 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);
```

Listing 7-3. Using a General-Purpose Timer as a Core Timer

```
/* Register Definitions */#define TMSTAT (0x1400)  /* GP Timer
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 */


/* Bit Definitions */#define TIMODEPWM   (0x00000001)

#define PRDCNT      (0x00000008)

#define IRQEN       (0x00000010)

#define TIM0EN      (0x00000100)


/* Main code section */

.global _main;

.section/pm seg_pmco;

_main:


/* Using PWM Out mode as a core timer */

ustat3 = TIMODEPWM|     /* PWM Out Mode */

 PRDCNT|       /* Count to end of period */

 IRQEN;
```

```
dm(TMOCTL) = ustat3;
R0 = 0x8000;
dm(TMOPRD) = R0;    /* Timer 0 period = 0x8000 */
R1 = 1;
dm(TMOW) = R1;      /* Timer 0 Pulse width = 1 */
R0 = TIMOEN;        /* enable timer 0 */
dm(TMSTAT) = R0;

/* Get start clock count */
R1 = EMUCLK;

// Wait until TIMOIRQ is set
// Alternatively, we could test GPTMR0I in IRPTL
r0=dm(TMSTAT);
btst r0 by 0;
if not sz jump (pc,2);
jump(pc,-3) (db);

/* Get end clock count */
R2=EMUCLK;
```

```
/* Subtract the start count from the end count
to obtain the number of cycles before the interrupt */
R4=R2-R1;

// R4 will be double the value of TM0PRD

_main.end: jump(pc,0);
```

A REGISTERS

The DSP 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:

When writing DSP programs, it is often necessary to set, clear, or test bits in the DSP'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/2126x/`include directory.

Many registers have reserved bits. When writing to a register, programs may only clear (write zero to) the register's reserved bits.

Control and Status System Registers

The DSP's Control And Status System registers determine how the processor core operates and indicate the status of many processor core operations. In the *ADSP-21160 SHARC DSP Instruction Set Reference*, these registers are referred to as System registers (`Sreg`), which are a subset of the DSP'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 A-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. For more information, see "I/O Processor Registers" on page A-51.

Table A-1. Control and Status Registers for the Processor Core

Register Name and Page Reference	Initialization After Reset
"Mode Control 1 Register (MODE1)" on page A-3	0x0000 0000
"Mode Mask Register (MMASK)" on page A-7	0x0020 0000
"Mode Control 2 Register (MODE2)" on page A-11	0x4200 0000
"Arithmetic Status Registers (ASTATx and ASTATy)" on page A-12	0x0000 0000
"Sticky Status Registers (STKYx and STKYy)" on page A-17	0x0540 0000
"User-Defined Status Registers (USTATx)" on page A-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 A-1 and Table A-3 provide bit information for the MODE1 register.

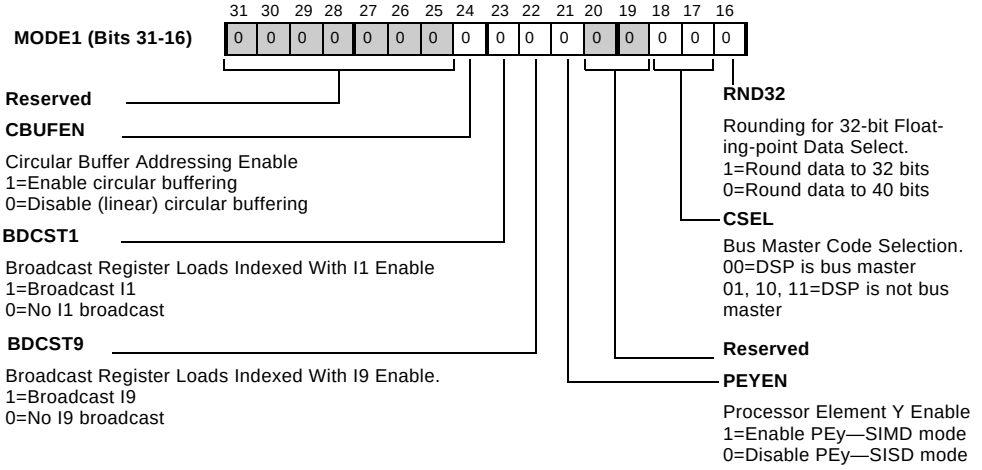


Figure A-1. Mode Control 1 Register (Upper)

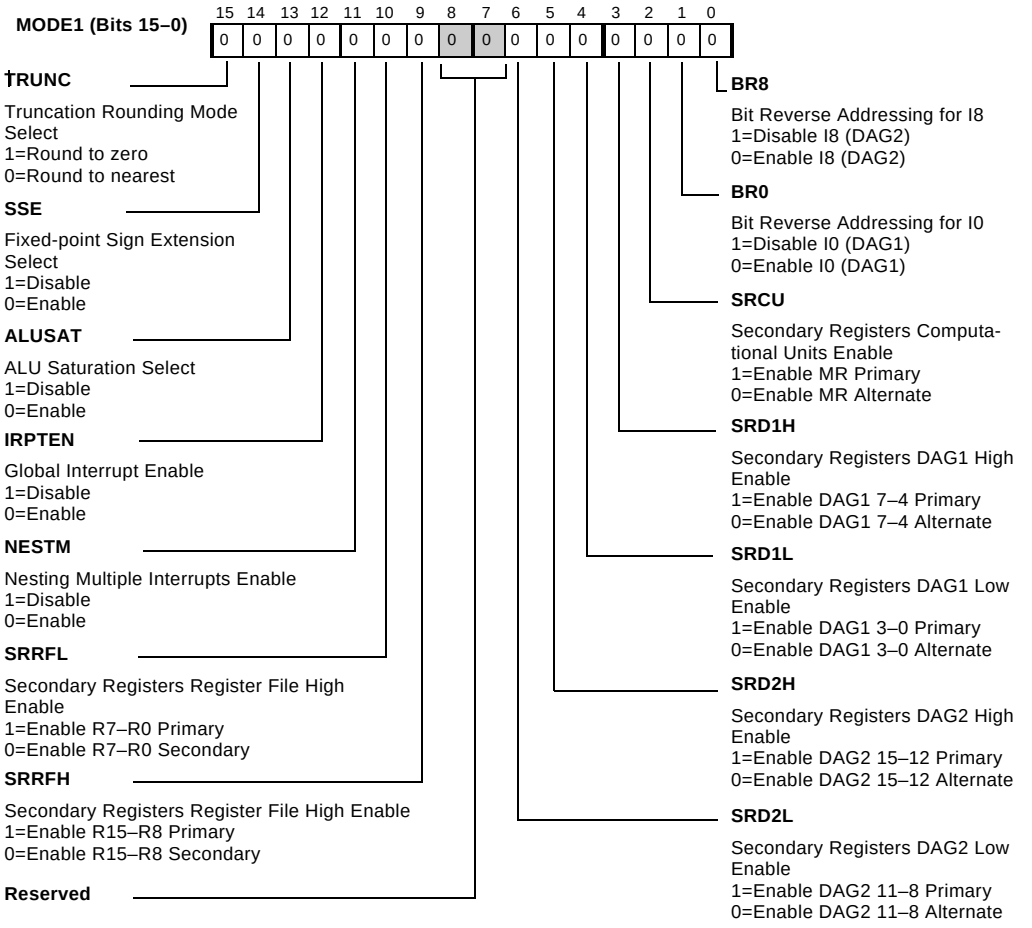


Figure A-2. Mode Control 1 Register (Lower)

Table A-2. Mode Control 1 Register (MODE1) Bit Definitions

Bit(s)	Name	Definition
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 A-2. Mode Control 1 Register (MODE1) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
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 DSP 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 DSP 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 Code Selection.** These bits indicate whether the DSP processor has control of the external bus as follows: 00 = DSP is bus master or 01, 10, 11 = DSP 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 A-2. Mode Control 1 Register (MODE1) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
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 DSP's status stack is pushed. This effectively disables different modes upon servicing an interrupt, or when executing a PUSH STS instruction.

The DSP's status stack will be pushed in two cases:

1. When executing a PUSH STS instruction explicitly in your code.
2. When an $\overline{\text{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 settings being enabled:

- 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.

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 will automatically disable SIMD when servicing any of the hardware interrupts mentioned above, or during any push of the status stack.

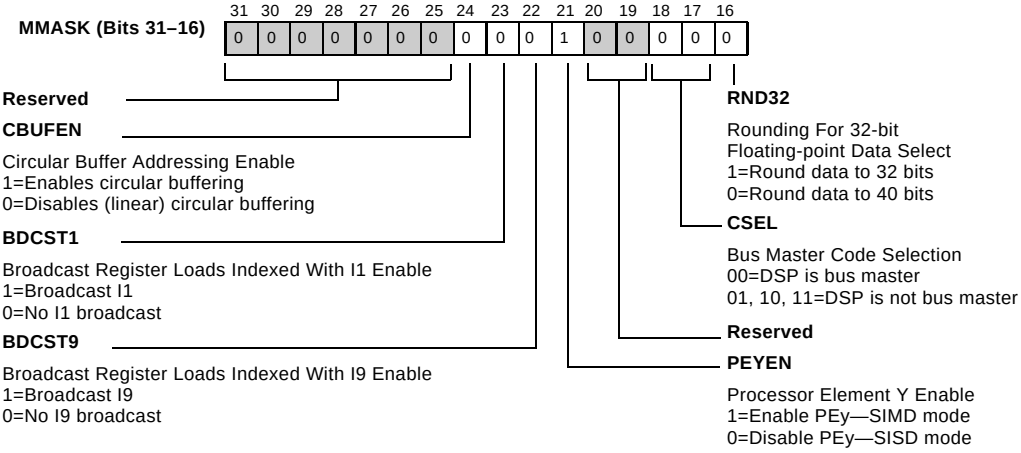


Figure A-3. MMASK Register (Upper Bits)

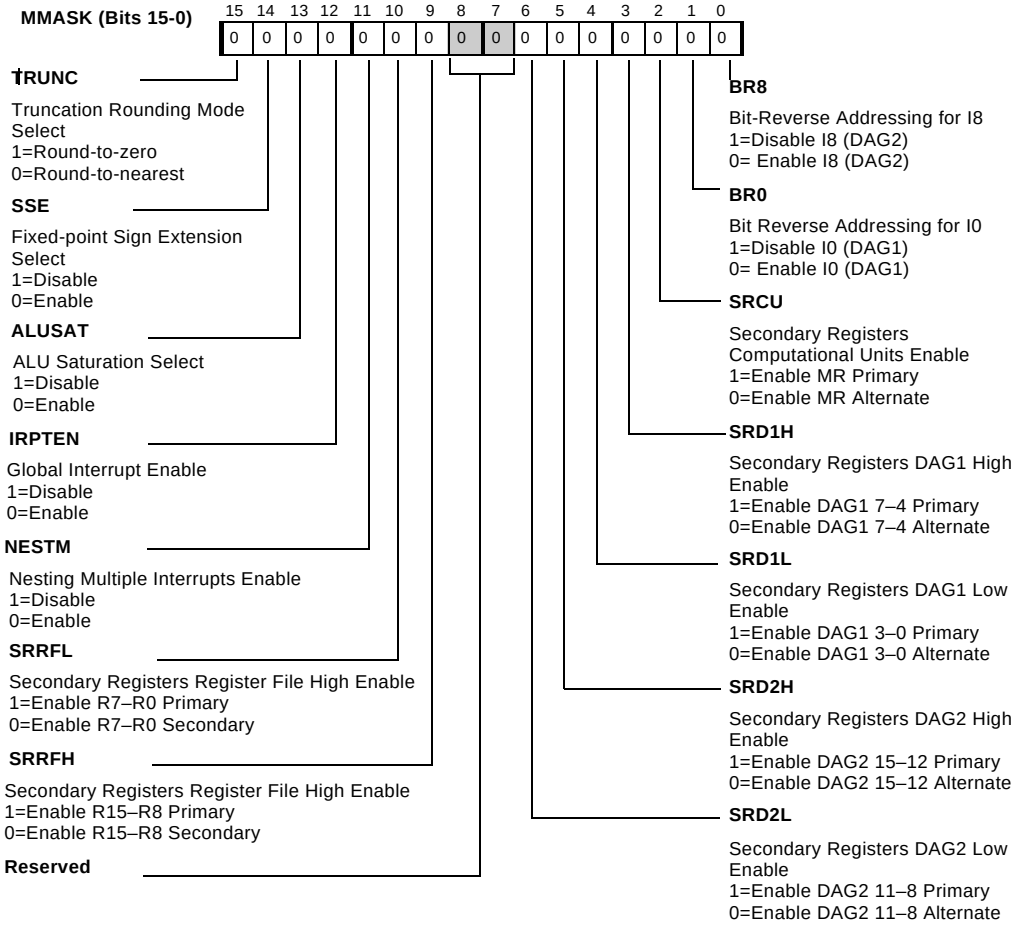


Figure A-4. MMASK Register (Lower Bits)

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 A-1 and Table A-3 provide bit information for the MODE2 register.

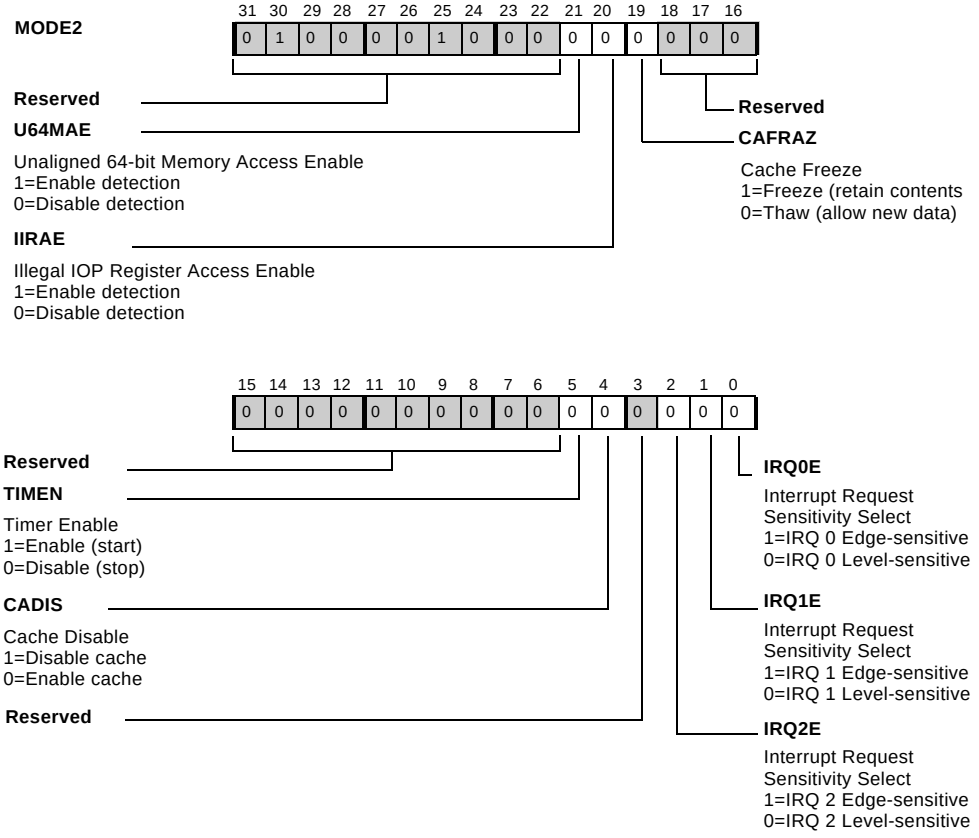


Figure A-5. MODE 2 Control Register

Table A-3. Mode Control 2 Register Bit Descriptions

Bits	Name	Definition
0	IRQ0E	$\overline{\text{IRQ0}}$ **Sensitivity Select.** Selects sensitivity for the flag configured as $\overline{\text{IRQ0}}$ as edge-sensitive (if set, = 1) or level-sensitive (if cleared, = 0).
1	IRQ1E	$\overline{\text{IRQ1}}$ **Sensitivity Select.** Selects sensitivity for the flag configured as $\overline{\text{IRQ1}}$ as edge-sensitive (if set, = 1) or level-sensitive (if cleared, = 0).
2	IRQ2E	$\overline{\text{IRQ2}}$ **Sensitivity Select.** Selects sensitivity for the flag configured as $\overline{\text{IRQ2}}$ 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 DSP 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 DSP flags an unaligned long word access by setting the U64MA bit in the STKYx register.
31–22	Reserved	

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 A-1 and Table A-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.

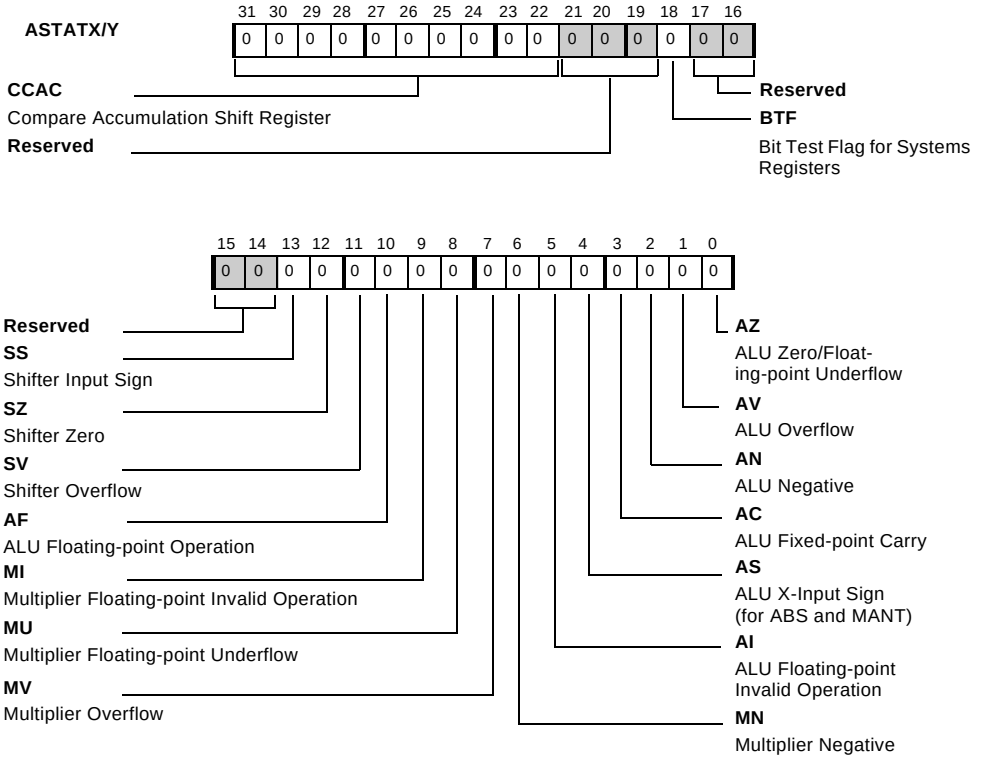


Figure A-6. ASTAT Register

Table A-4. ASTATx and ASTATy Register Bit Descriptions

Bits	Name	Definition
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 DSP 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 DSP 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 DSP 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 DSP clears AC during fixed-point logic operations: PASS, MIN, MAX, COMP, ABS, and CLIP. The ALU reads the AC flag for 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 A-4. ASTATx and ASTATy Register Bit Descriptions (Cont'd)

Bits	Name	Definition
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 DSP 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 DSP sets MV and MVS in the STKYx/y register if the rounded result overflows (unbiased exponent > 127). For fixed-point results, the DSP 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 DSP 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 A-4. ASTATx and ASTATy Register Bit Descriptions (Cont'd)

Bits	Name	Definition
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 DSP 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 DSP 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 DSP 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 DSP 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 DSP 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 DSP also sets SZ if the shifter operation performs a bit test on a bit outside of the 32-bit fixed-point field.

Table A-4. ASTATx and ASTATy Register Bit Descriptions (Cont'd)

Bits	Name	Definition
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 DSP sets BTF when the bit(s) in a System register and value in the Bit Tst instruction match. The DSP 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 DSP 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. Table A-6 lists bits 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 DSP sets a STKY bit in response to a condition. For example, the DSP sets the AUS bit in the STKY register when an ALU underflow set AZ in the ASTAT register. The DSP 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 DSP 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.

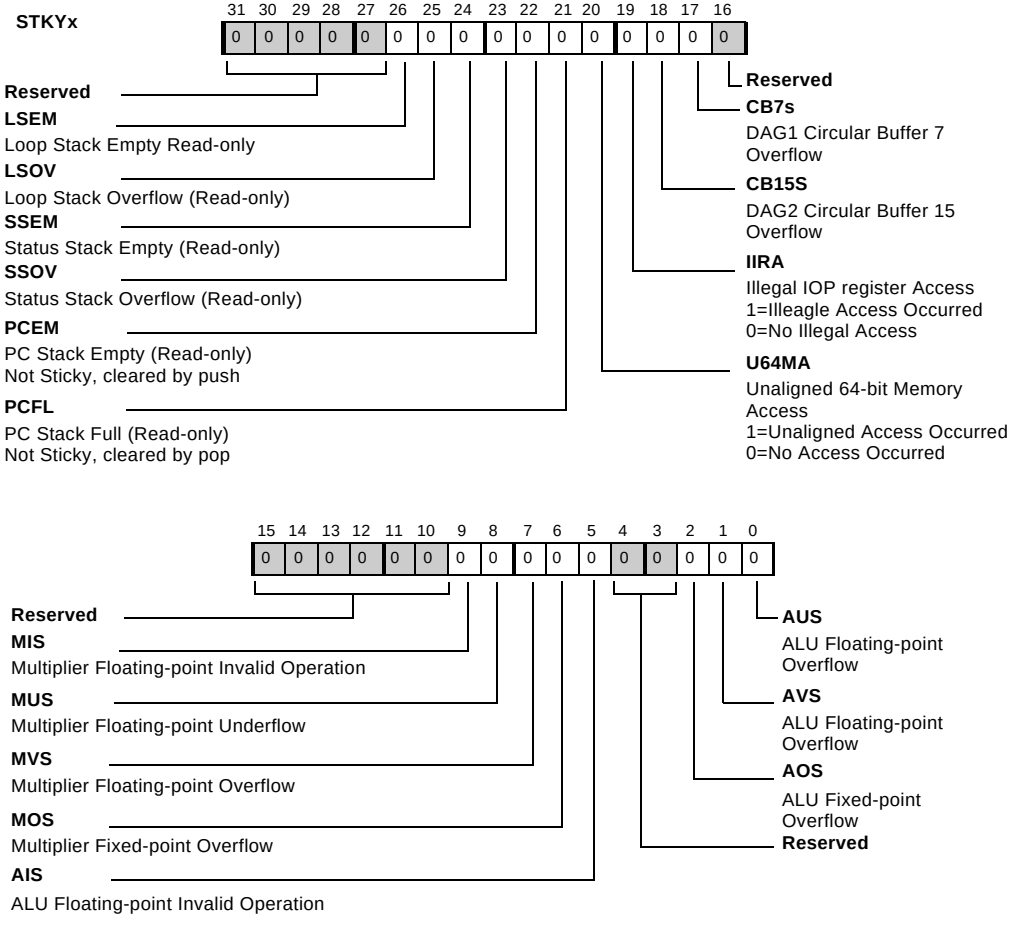


Figure A-7. STKYx Register

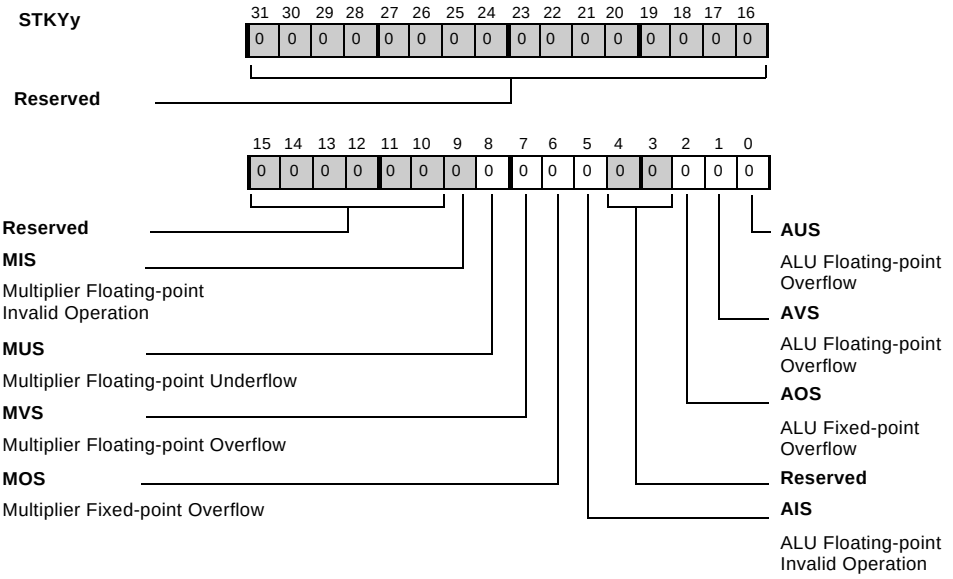


Figure A-8. STKYy Register

Table A-5. STKYx and STKYy Registers Bit Descriptions

Bits	Name	Definition: √ 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 A-14.	√
1	AVS	**ALU Floating-Point Overflow.** A sticky indicator for the ALU AV bit. For more information, see "AV" on page A-14.	√
2	AOS	**ALU Fixed-Point Overflow.** A sticky indicator for the ALU AV bit. For more information, see "AV" on page A-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 A-15.	√
6	MOS	**Multiplier Fixed-Point Overflow.** A sticky indicator for the multiplier MV bit. For more information, see "MV" on page A-15.	√

Table A-5. STKYx and STKYy Registers Bit Descriptions (Cont'd)

Bits	Name	Definition: √ shows bits in both STKYx/y; × shows bits in STKYx only	
7	MVS	**Multiplier Floating-Point Overflow.** A sticky indicator for the multiplier MV bit. For more information, see "MV" on page A-15.	√
8	MUS	**Multiplier Floating-Point Underflow.** A sticky indicator for the multiplier MU bit. For more information, see "MU" on page A-16.	√
9	MIS	**Multiplier Floating-Point Invalid Operation.** A sticky indicator for the multiplier MI bit. For more information, see "MI" on page A-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.	×
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.	×

Table A-5. STKYx and STKYy Registers Bit Descriptions (Cont'd)

Bits	Name	Definition: √ shows bits in both STKYx/y; × shows bits in STKYx only	
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 DSP'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.

Table A-6. Processing Element Registers

Register Name and Page Reference	Initialization After Reset
"Data File Data Registers (Rx, Fx, Sx)" on page A-22	Undefined
"Multiplier Results Registers (MRFx, MRBx)" on page A-22	Undefined
"Program Memory Bus Exchange Register (PX)" on page A-23	Undefined

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 DSP'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 DSP's multipliers has a primary or foreground (MRF) register and alternate or background (MRB) results register. 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.

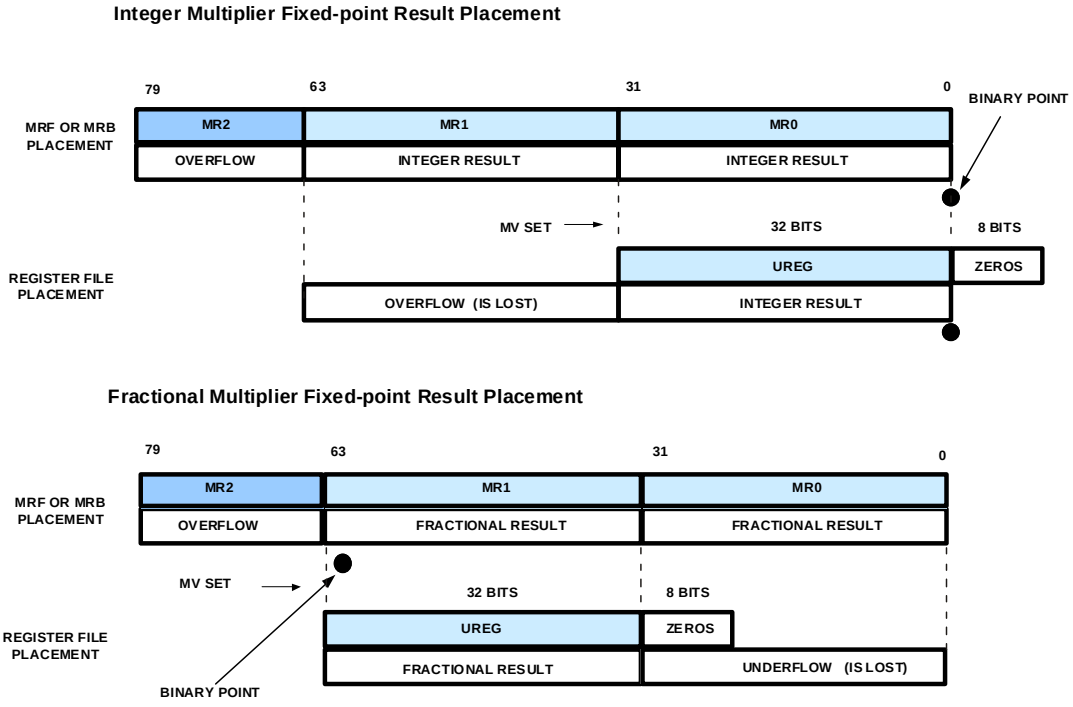


Figure A-9. MRFx and MRBx Registers

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 DSP's program sequencer registers direct the execution of instructions. These registers include support for the:

- Instruction pipeline
- Program and loop stacks
- Timer
- Interrupt mask and latch

Table A-7. Program Sequencer Registers

Register	Initialization After Reset
"Interrupt Latch Register (IRPTL)" on page A-25	0x0000 0000 (cleared)
"Interrupt Mask Register (IMASK)" on page A-30	0x0000 0003
"Interrupt Mask Pointer Register (IMASKP)" on page A-35	0x0000 0000 (cleared)
"Interrupt Register (LIRPTL)" on page A-42	0x0000 0000 (cleared)

Table A-8. Program Counter Registers

Register	Initialization After Reset
"Program Counter Register (PC)" on page A-47	Undefined
"Program Counter Stack Register (PCSTK)" on page A-48	Undefined
"Program Counter Stack Pointer Register (PCSTKP)" on page A-48	Undefined
"Fetch Address Register (FADDR)" on page A-48	Undefined
"Decode Address Register (DADDR)" on page A-48	Undefined
"Loop Address Stack Register (LADDR)" on page A-49	Undefined
"Current Loop Counter Register (CURLCNTR)" on page A-49	Undefined
"Loop Counter Register (LCNTR)" on page A-49	Undefined

Table A-8. Program Counter Registers (Cont'd)

Register	Initialization After Reset
"Timer Period Register (TPERIOD)" on page A-50	Undefined
"Timer Count Register (TCOUNT)" on page A-50	Undefined

Interrupt Latch Register (IRPTL)

The IRPTL register is a non memory-mapped, universal, system register (*Ureg* and *Sreg*). The reset value for this register is 0x0000 0000. The IRPTL register indicates latch status for interrupts. Figure A-10 and Table A-9 provide bit definitions for the IRPTL register.

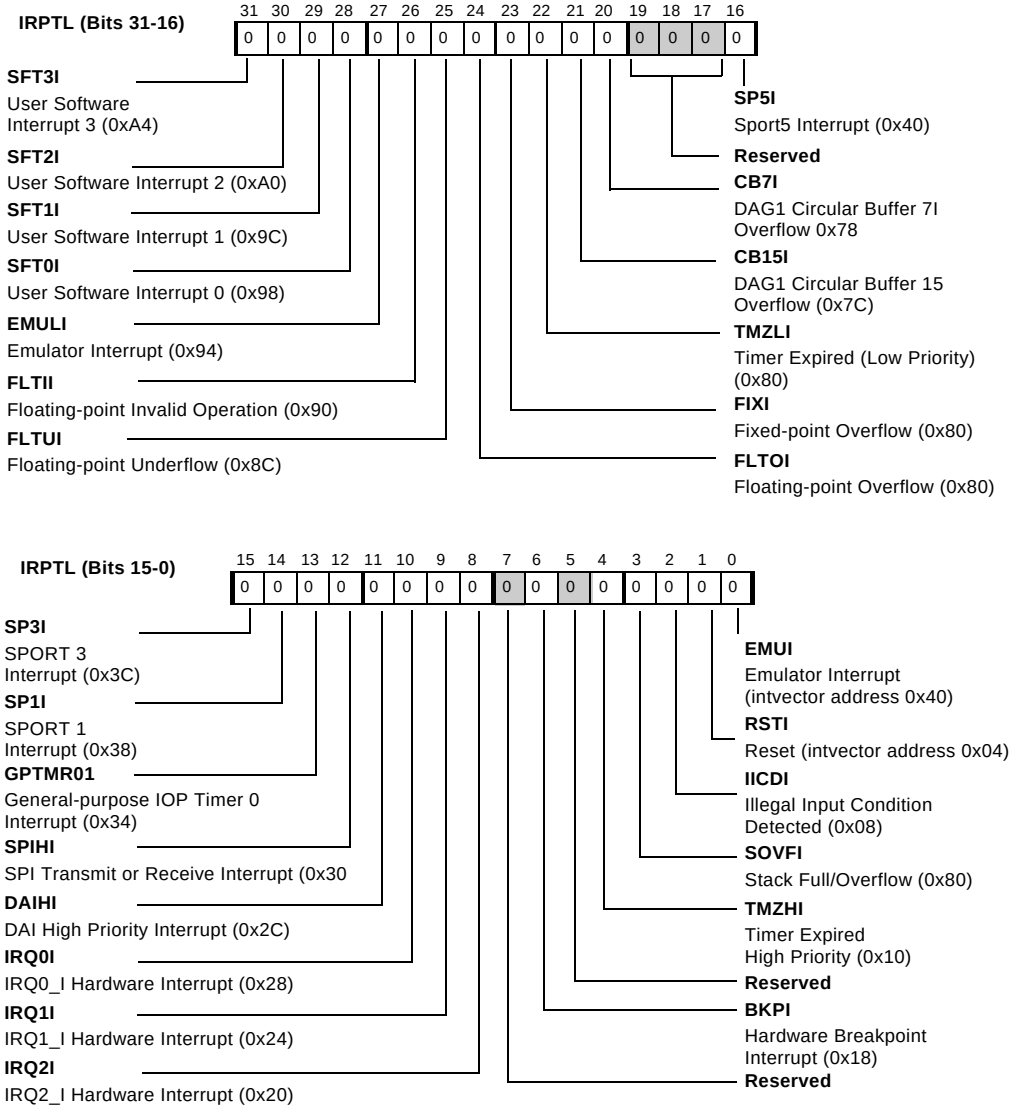


Figure A-10. IRPTL Register

Table A-9. IRPTL Register Bit Descriptions

Bits	Name	Definition
0	EMUI	**Emulator Interrupt.** Indicates if an EMUI is latched and is pending (if set, = 1) or no EMUI is pending (if cleared, = 0). An EMUI occurs on reset and when an external device asserts the $\overline{\text{EMU}}$ pin.
1	RSTI	**Reset Interrupt.** Indicates if an RSTI is latched and is pending (if set, = 1) or no RSTI is pending (if cleared, = 0). An RSTI occurs on reset as an external device asserts the $\overline{\text{RESET}}$ pin.
2	IICDI	**Illegal Input Condition Detected Interrupt.** Indicates if an IICDI is latched and is pending (if set, = 1) or no IICDI is pending (if cleared, = 0). An IICDI occurs when a TRUE results from the logical Or'ing of the Illegal I/O Processor Register Access (IIRA) and Unaligned 64-bit Memory Access bits in the STKYx registers.
3	SOVFI	**Stack Overflow/Full Interrupt.** Indicates if a SOVFI is latched and is pending (if set, = 1) or no SOVFI is pending (if cleared, = 0). An SOVFI occurs when a stack in the program sequencer overflows or is full. For more information, see "PCFL" on page A-20, SSOV bit "SSOV" on page A-20, and "LSOV" on page A-21.
4	TMZHI	**Timer Expired High Priority.** Indicates if a TMZHI is latched and is pending (if set, = 1) or TMZHI is not pending (if cleared, = 0). A TMZHI occurs when the timer decrements to zero. Note that this event also triggers a TMZLI. The timer operations are controlled as follows: • The TCOUNT register contains the timer counter. The timer decrements the TCOUNT register each clock cycle. • The TPERIOD value specifies the frequency of timer interrupts. The number of cycles between interrupts is TPERIOD + 1. The maximum value of TPERIOD is $2^{32} - 1$. • The TIMEN bit in the MODE2 register starts and stops the timer. Since the timer expired event (TCOUNT decrements to zero) generates two interrupts, TMZHI and TMZLI, programs should unmask the timer interrupt with the desired priority and leave the other one masked.
5	Reserved	
6	BKPI	**Hardware Breakpoint Interrupt.** Indicates if an BKPI is latched and is pending (if set, = 1) or no BKPI is pending (if cleared, = 0).

Table A-9. IRPTL Register Bit Descriptions (Cont'd)

Bits	Name	Definition
7	Reserved	
8	$\overline{\text{IRQ2}}$I	$\overline{\text{IRQ2}}$ **Hardware Interrupt.** Indicates if an IRQ2I is latched and is pending (if set, = 1) or no IRQ2I is pending (if cleared, = 0). An IRQ2I occurs when an external device asserts the FLG2 pin configured as $\overline{\text{IRQ2}}$.
9	$\overline{\text{IRQ1}}$I	$\overline{\text{IRQ1}}$ **Hardware Interrupt.** Indicates if an $\overline{\text{IRQ1}}$I is latched and is pending (if set, = 1) or no $\overline{\text{IRQ1}}$I is pending (if cleared, = 0). An $\overline{\text{IRQ1}}$I occurs when an external device asserts the FLG1 pin configured as $\overline{\text{IRQ1}}$.
10	$\overline{\text{IRQ0}}$I	$\overline{\text{IRQ0}}$ **Hardware Interrupt.** Indicates if an $\overline{\text{IRQ0}}$I is latched and is pending (if set, = 1) or no IRQ0I is pending (if cleared, = 0). An IRQ0I occurs when an external device asserts the FLG0 pin configured as $\overline{\text{IRQ0}}$.
11	DAIHI	**DAI High Priority Interrupt.** Indicates if a DAI interrupt is latched and is pending (if set, = 1) or no DAI interrupt is pending (if cleared, = 0). This is the higher priority option.
12	SPIHI	**SPI Transmit or Receive Interrupt.** Indicates if a SPIHI is latched and is pending (if set, = 1) or no SPIHI is pending (if cleared, = 0). This is the higher priority option.
13	GPTMR0I	**General-Purpose IOP Timer 0 Interrupt.** Indicates if a GPTMR0I is latched and is pending (if set, = 1) or no GPTMR0I is pending (if cleared, = 0).
14	SP1I	**SPORT 1 Interrupt.** Indicates if an SP1I interrupt is latched and is pending (if set, = 1), or no SP1I is pending (if cleared, = 0). An SP1I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP1A/TXSP1A, RXSP1B/TXSP1B.
15	SP3I	**SPORT 3 Interrupt.** Indicates if an SP3I interrupt is latched and is pending (if set, = 1), or no SP3I is pending (if cleared, = 0). An SP3I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP3A/TXSP3A, RXSP3B/TXSP3B.
16	SP5I	**SPORT 5 Interrupt.** Indicates if an SP5I interrupt is latched and is pending (if set, = 1), or no SP5I is pending (if cleared, = 0). An SP5I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP5A/TXSP5A, RXSP5B/TXSP5B.

Table A-9. IRPTL Register Bit Descriptions (Cont'd)

Bits	Name	Definition
19–17	Reserved	
20	CB7I	**DAG1 Circular Buffer 7 Overflow Interrupt.** Indicates if a CB7I is latched and is pending (if set, = 1) or no CB7I interrupt is pending (if cleared, = 0). A circular buffer overflow occurs when the DAG circular buffering operation increments the I register past the end of the buffer. For more information, see "CB7S" on page A-20.
21	CB15I	**DAG2 Circular Buffer 15 Overflow Interrupt.** Indicates if a CB15I is latched and is pending (if set, = 1) or no CB15I is pending (if cleared, = 0). A circular buffer overflow occurs when the DAG circular buffering operation increments the I register past the end of the buffer. For more information, see "CB15S" on page A-20.
22	TMZLI	**Timer Expired (Low Priority) Interrupt.** Indicates if a TMZLI is latched and is pending (if set, = 1) or no TMZLI is pending (if cleared, = 0). For more information, see "TMZHI" on page A-27.
23	FIXI	**Fixed-Point Overflow Interrupt.** Indicates if a FIXI is latched and is pending (if set, = 1) or no FIXI is pending (if cleared, = 0). For more information, see "AOS" on page A-19.
24	FLTOI	**Floating-Point Overflow Interrupt.** Indicates if a FLTOI is latched and is pending (if set, = 1) or no FLTOI is pending (if cleared, = 0).
25	FLTUI	**Floating-Point Underflow Interrupt.** Indicates if a FLTUI is latched and is pending (if set, = 1) or no FLTUI is pending (if cleared, = 0).
26	FLTII	**Floating-Point Invalid Operation Interrupt.** This bit indicates if a FLTII is latched and is pending (if set, = 1) or no FLTII is pending (if cleared, = 0). For more information, see "AIS" on page A-19.
27	EMULI	**Emulator (Lower Priority) Interrupt.** Indicates if an EMUI is latched and is pending (if set, = 1) or no EMULI is pending (if cleared, = 0). An EMULI occurs on reset and when an external device asserts the $\overline{\text{EMU}}$ pin. This interrupt has a lower priority than EMUI, but higher priority than software interrupts.
28	SFT0I	**User Software Interrupt 0.** Indicates if a SFT0I is latched and is pending (if set, = 1) or no SFT0I is pending (if cleared, = 0). An SFT0I occurs when a program sets (= 1) this bit.

Table A-9. IRPTL Register Bit Descriptions (Cont'd)

Bits	Name	Definition
29	SFT1I	**User Software Interrupt 1.** Indicates if a SFT1I is latched and is pending (if set, = 1) or no SFT1I is pending (if cleared, = 0). For details, see SFT0I bit description.
30	SFT2I	**User Software Interrupt 2.** Indicates if a SFT2I is latched and is pending (if set, = 1) or no SFT2I is pending (if cleared, = 0). For details, see SFT0I bit description.
31	SFT3I	**User Software Interrupt 3.** Indicates if a SFT3I is latched and is pending (if set, = 1) or no SFT3I is pending (if cleared, = 0). For details, see SFT0I bit description.

Interrupt Mask Register (IMASK)

The IMASK register is a non-memory-mapped, universal, system register (*Ureg* and *Sreg*). The reset value for this register is 0x0000 0003. Each bit in the IMASK register corresponds to a bit with the same name in the IRPTL registers. The bits in IMASK unmask (enable if set, = 1) or mask (disable if cleared, = 0) the interrupts that are latched in the IRPTL register. Except for $\overline{\text{RSTI}}$ and EMUI, all interrupts are maskable.

When IMASK masks an interrupt, the masking disables the DSP's response to the interrupt. The IRPTL register still latches an interrupt even when masked, and the DSP responds to that latched interrupt if it is later unmasked. Table A-10 and Figure A-11 provide bit definitions for the IMASK register.

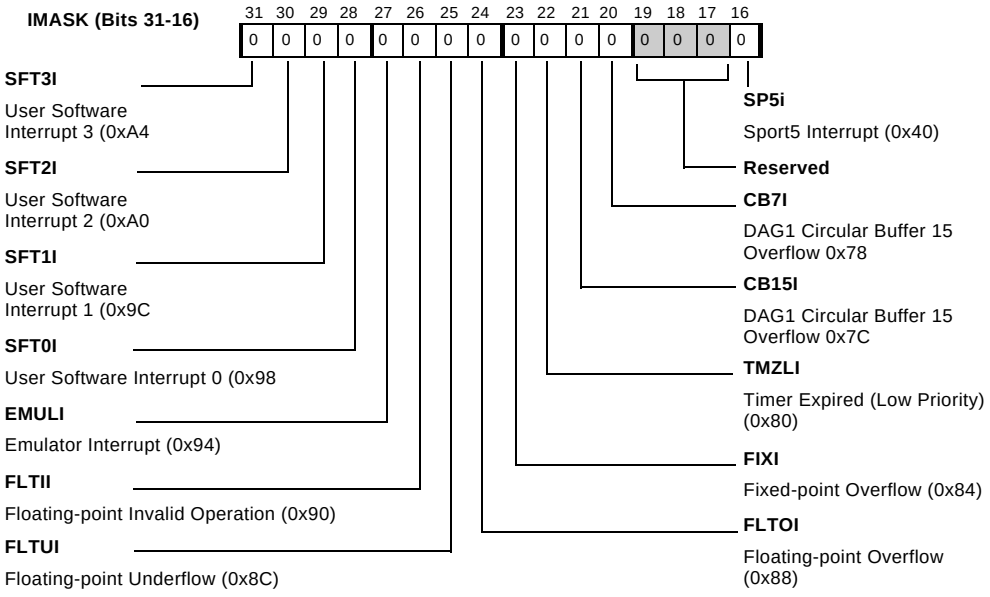


Figure A-11. IMASK Register (Upper Bits)

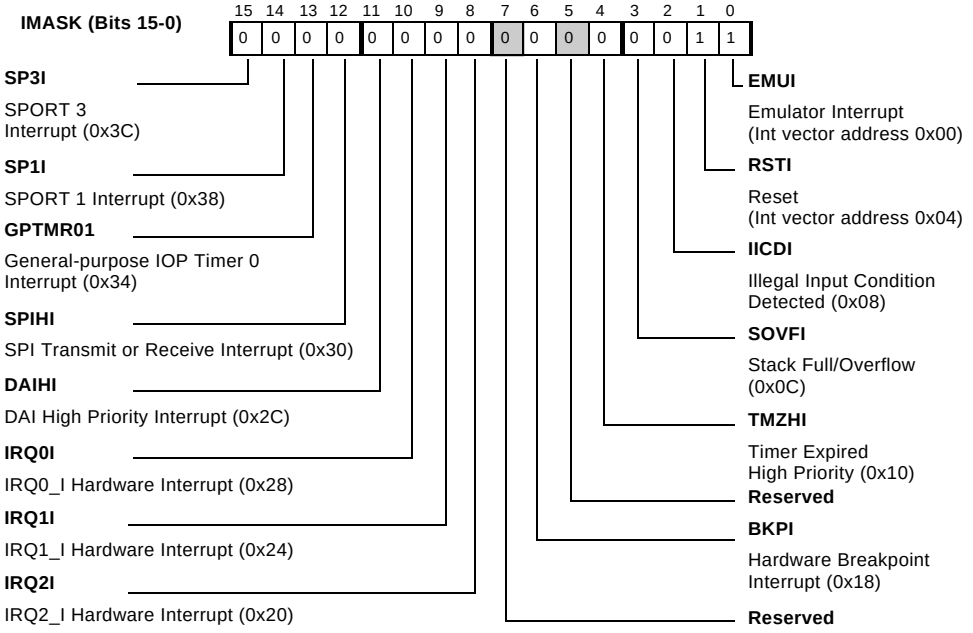


Figure A-12. IMASK Register (Lower Bits)

Table A-10. IMASK Register Bit Descriptions

Bits	Name	Definition
0	EMUI	**Emulator Interrupt.** This bit is set to 1 (unmasked). An EMUI occurs on reset and when an external device asserts the $\overline{\text{EMU}}$ pin.
1	RSTI	**Reset Interrupt.** This bit is set to 1 (unmasked). An RSTI occurs on reset as an external device asserts the $\overline{\text{RESET}}$ pin.
2	IICDI	**Illegal Input Condition Detected Interrupt.** Unmasks the IICDI interrupt (if set, = 1) or masks (if cleared, = 0). An IICDI occurs when a TRUE results from the logical ORing of the Illegal I/O Processor Register Access (IIRA) and Unaligned 64-bit Memory Access bits in the STKYx registers.

Table A-10. IMASK Register Bit Descriptions (Cont'd)

Bits	Name	Definition
3	SOVFI	**Stack Overflow/Full Interrupt.** Unmasks the SOVFI interrupt (if set, = 1) or masks the SOVFI interrupt (if cleared, = 0). An SOVFI occurs when a stack in the program sequencer overflows or is full. For more information, see "PCFL" on page A-20, SSOV bit "SSOV" on page A-20, and "LSOV" on page A-21.
4	TMZHI	**Timer Expired High Priority.** Unmasks the TMZHI interrupt (if set, = 1) or masks the TMZHI interrupt (if cleared, = 0). A TMZHI occurs when the timer decrements to zero. Note that this event also triggers a TMZLI. The timer operations are controlled as follows: • The TCOUNT register contains the timer counter. The timer decrements the TCOUNT register each clock cycle. • The TPERIOD value specifies the frequency of timer interrupts. The number of cycles between interrupts is TPERIOD + 1. The maximum value of TPERIOD is $2^{32} - 1$. • The TIMEN bit in the MODE2 register starts and stops the timer. Since the timer expired event (TCOUNT decrements to zero) generates two interrupts, TMZHI and TMZLI, programs should unmask the timer interrupt with the desired priority and leave the other one masked.
5	Reserved	
6	BKPI	**Hardware Breakpoint Interrupt.** Unmasks the BKPI interrupt (if set, = 1) or masks the BKPI interrupt (if cleared, = 0).
7	Reserved	
8	IRQ2I	$\overline{\text{IRQ2}}$ **Hardware Interrupt.** Unmasks the $\overline{\text{IRQ2I}}$ interrupt (if set, = 1) or masks the interrupt (if cleared, = 0). An $\overline{\text{IRQ2I}}$ occurs when an external device asserts the FLG2 pin configured as $\overline{\text{IRQ2}}$.
9	IRQ1I	$\overline{\text{IRQ1}}$ **Hardware Interrupt.** Unmasks the $\overline{\text{IRQ1I}}$ interrupt (if set, = 1) or masks the $\overline{\text{IRQ1I}}$ interrupt (if cleared, = 0). An $\overline{\text{IRQ1I}}$ occurs when an external device asserts the FLG1 pin configured as $\overline{\text{IRQ1}}$.
10	IRQ0I	$\overline{\text{IRQ0}}$ **Hardware Interrupt.** Unmasks the IRQ0I interrupt (if set, = 1) or masks the $\overline{\text{IRQ0I}}$ interrupt (if cleared, = 0). An $\overline{\text{IRQ0I}}$ occurs when an external device asserts the FLG0 pin configured as $\overline{\text{IRQ0}}$.

Table A-10. IMASK Register Bit Descriptions (Cont'd)

Bits	Name	Definition
11	DAIHI	**DAI High Priority Interrupt.** Unmasks the DAIHI interrupt (if set, = 1) or masks the DAIHI interrupt (if cleared, = 0). This is the higher priority option.
12	SPIHI	**SPI Transmit or Receive Interrupt.** Unmasks the SPIHI interrupt (if set, = 1) or masks the SPIHI interrupt (if cleared, = 0). This is the higher priority option.
13	GPTMR0I	**General-Purpose IOP Timer 0 Interrupt.** Unmasks the GPTMR0I interrupt (if set, = 1) or masks the GPTMR0I interrupt (if cleared, = 0).
14	SP1I	**SPORT 1 Interrupt.** Unmasks the SP1I interrupt (if set, = 1) or masks the SP1I interrupt (if cleared, = 0). An SP1I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP1A/TXSP1A, or RXSP1B/TXSP1B.
15	SP3I	**SPORT 3 Interrupt.** Unmasks the SP3I interrupt (if set, = 1) or masks the SP3I interrupt (if cleared, = 0). An SP3I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP3A/TXSP3A, or RXSP3B/TXSP3B.
16	SP5I	**SPORT 5 Interrupt.** Unmasks the SP5I interrupt (if set, = 1) or masks the SP5I interrupt (if cleared, = 0). An SP5I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP5A/TXSP5A, or RXSP5B/TXSP5B.
19–17	Reserved	
20	CB7I	**DAG1 Circular Buffer 7 Overflow Interrupt.** Unmasks the CB7I interrupt (if set, = 1) or masks the CB7I interrupt (if cleared, = 0). A circular buffer overflow occurs when the DAG circular buffering operation increments the I register past the end of the buffer. For more information, see "CB7S" on page A-20.
21	CB15I	**DAG2 Circular Buffer 15 Overflow Interrupt.** Unmasks the CB15I interrupt (if set, = 1) or masks the CB15I interrupt (if cleared, = 0). A circular buffer overflow occurs when the DAG circular buffering operation increments the I register past the end of the buffer. For more information, see "CB15S" on page A-20.
22	TMZLI	**Timer Expired (Low Priority) Interrupt.** Unmasks the TMZLI interrupt (if set, = 1) or masks the TMZLI interrupt (if cleared, = 0). For more information, see "TMZHI" on page A-27.

Table A-10. IMASK Register Bit Descriptions (Cont'd)

Bits	Name	Definition
23	FIXI	**Fixed-Point Overflow Interrupt.** Unmasks the FIXI interrupt (if set, = 1) or masks the FIXI interrupt (if cleared, = 0). For more information, see "AOS" on page A-19.
24	FLTOI	**Floating-Point Overflow Interrupt.** Unmasks the FLTOI interrupt (if set, = 1) or masks the FLTOI interrupt (if cleared, = 0).
25	FLTUI	**Floating-Point Underflow Interrupt.** Unmasks the FLTUI interrupt (if set, = 1) or masks the FLTUI interrupt (if cleared, = 0).
26	FLTII	**Floating-Point Invalid Operation Interrupt.** Unmasks the FLTII interrupt (if set, = 1) or masks the FLTII interrupt (if cleared, = 0). For more information, see "AIS" on page A-19.
27	EMULI	**Emulator (Lower Priority) Interrupt.** Unmasks the EMULI interrupt (if set, = 1) or masks the EMULI interrupt (if cleared, = 0). An EMULI occurs on reset and when an external device asserts the $\overline{\text{EMU}}$ pin. This interrupt has a lower priority than EMUI, but higher priority than software interrupts.
28	SFT0I	**User Software Interrupt 0.** Unmasks the SFT0I interrupt (if set, = 1) or masks the SFT0I interrupt (if cleared, = 0). An SFT0I occurs when a program sets (= 1) this bit.
29	SFT1I	**User Software Interrupt 1.** Unmasks the SFT1I interrupt (if set, = 1) or masks the SFT1I interrupt (if cleared, = 0).
30	SFT2I	**User Software Interrupt 2.** Unmasks the SFT2I interrupt (if set, = 1) or masks the SFT2I interrupt (if cleared, = 0).
31	SFT3I	**User Software Interrupt 3.** Unmasks the SFT3I interrupt (if set, = 1) or masks the SFT3I interrupt (if cleared, = 0).

Interrupt Mask Pointer Register (IMASKP)

The `IMASKP` register is a non-memory-mapped, universal, system register (*`Ureg`* and *`Sreg`*). The reset value for this register is 0x0000 0000. Each bit in the `IMASKP` register corresponds to a bit with the same name in the `IRPTL` registers. The `IMASKP` register field descriptions are described in Figure A-11, Figure A-12, and Table A-10.

This register supports an interrupt nesting scheme that lets higher priority events interrupt an ISR and keeps lower priority events from interrupting.

When interrupt nesting is enabled, the bits in the IMASKP register mask interrupts having lower priority than the interrupt that is currently being serviced. Other bits in this register unmask interrupts having higher priority than the interrupt that is currently being serviced. Interrupt nesting is enabled using NESTM in the MODE1 register. The IRPTL register latches a lower priority interrupt even when masked, and the DSP responds to that latched interrupt if it is later unmasked.

When interrupt nesting is disabled (NESTM = 0 in the MODE1 register), the bits in the IMASKP register mask all interrupts while an interrupt is currently being serviced. The IRPTL register still latches these interrupts even when masked, and the DSP responds to the highest priority latched interrupt after servicing the current interrupt. For more information, see “NESTM” on page A-6.

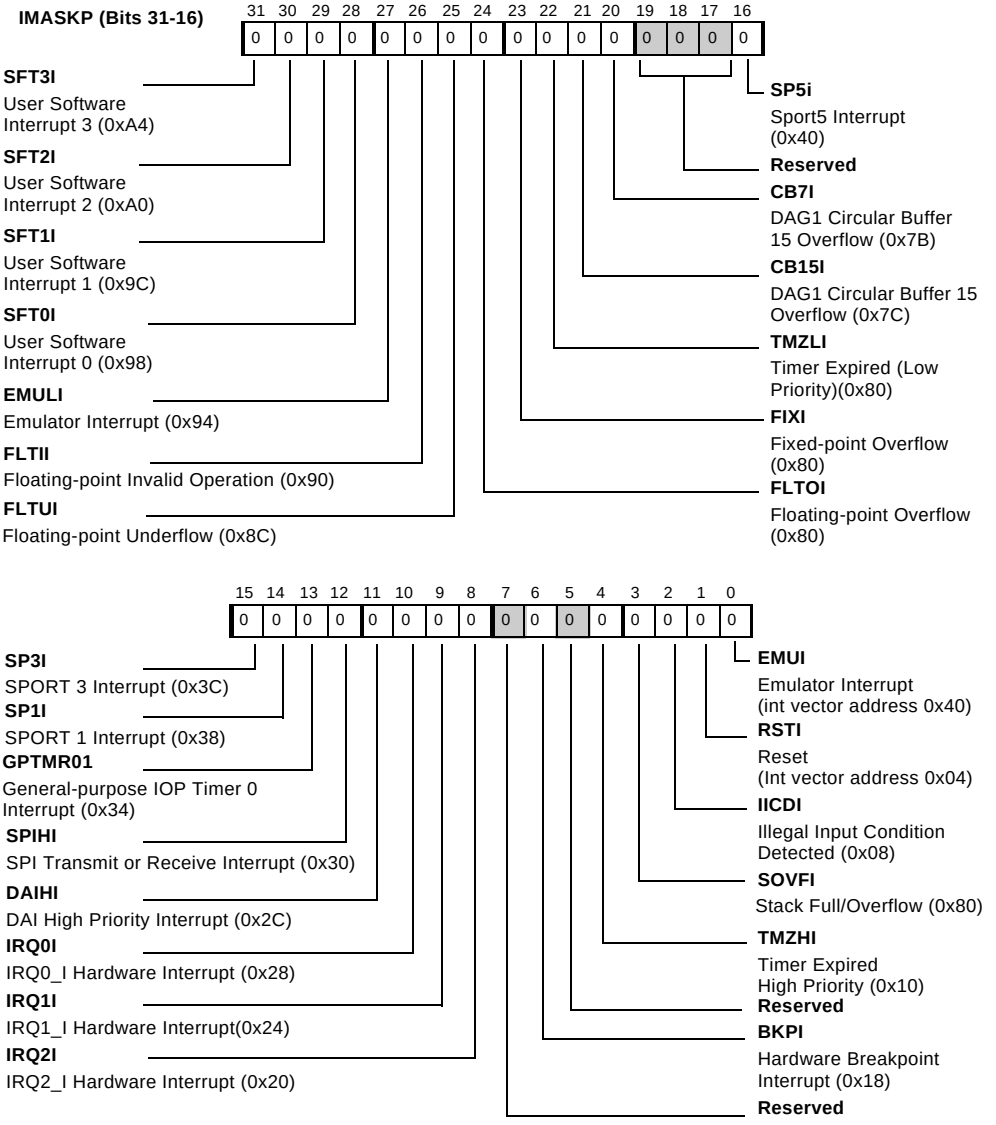


Figure A-13. IMASKP Register

Table A-11. IMASKP Register Bit Descriptions

Bits	Name	Definition
0	EMUI	**Emulator Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the EMUI interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). An EMUI occurs on reset and when an external device asserts the $\overline{\text{EMU}}$ pin.
1	RSTI	**Reset Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the RSTI interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). An RSTI occurs on reset as an external device asserts the $\overline{\text{RESET}}$ pin.
2	IICDI	**Illegal Input Condition Detected Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the IICDI interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). An IICDI occurs when a TRUE results from the logical ORing of the Illegal I/O Processor Register Access (IIRA) and Unaligned 64-bit Memory Access bits in the STKYx registers.
3	SOVFI	**Stack Overflow/Full Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the SOVFI interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). A SOVFI occurs when a stack in the program sequencer overflows or is full. For more information, see "PCFL" on page A-20, SSOV bit "SSOV" on page A-20, and "LSOV" on page A-21.
4	TMZHI	**Timer Expired High Priority.** When the DSP is servicing another interrupt, this bit indicates if the TMZHI interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). A TMZHI occurs when the timer decrements to zero. Note that this event also triggers a TMZLI. Timer operations are controlled as follows: • The TCOUNT register contains the timer counter. The timer decrements the TCOUNT register each clock cycle. • The TPERIOD value specifies the frequency of timer interrupts. The number of cycles between interrupts is TPERIOD + 1. The maximum value of TPERIOD is $2^{32} - 1$. • The TIMEN bit in the MODE2 register starts and stops the timer. Since the timer expired event (TCOUNT decrements to zero) generates two interrupts, TMZHI and TMZLI, programs should unmask the timer interrupt with the desired priority and leave the other one masked.

Table A-11. IMASKP Register Bit Descriptions (Cont'd)

Bits	Name	Definition
5	Reserved	
6	BKPI	**Hardware Breakpoint Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the BKPI interrupt is unmasked (if set, = 1) or masked (if cleared, = 0).
7	Reserved	
8	IRQ2I	$\overline{\text{IRQ2}}$ **Hardware Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the IRQ2I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). An IRQ2I occurs when an external device asserts the FLG2 pin configured as $\overline{\text{IRQ2}}$.
9	IRQ1I	$\overline{\text{IRQ1}}$ **Hardware Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the IRQ1I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). An IRQ1I occurs when an external device asserts the FLG1 pin configured as $\overline{\text{IRQ1}}$.
10	IRQ0I	$\overline{\text{IRQ0}}$ **Hardware Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the IRQ0I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). An IRQ0I occurs when an external device asserts the FLG0 pin configured as $\overline{\text{IRQ0}}$.
11	DAIHI	**DAI High Priority Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the DAIHI interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). This is the higher priority option.
12	SPIHI	**SPI Transmit or Receive Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the SPIHI interrupt is unmasked (if set, = 1) or the SPIHI interrupt is masked (if cleared, = 0). This is the higher priority option.
13	GPTMR0I	**General-Purpose IOP Timer 0 Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the GPTMR0I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0).
14	SP1I	**SPORT 1 Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the SP1I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). An SP1I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP1A/TXSP1A, or RXSP1B/TXSP1B.

Table A-11. IMASKP Register Bit Descriptions (Cont'd)

Bits	Name	Definition
15	SP3I	**SPORT 3 Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the SP3I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). An SP3I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP3A/TXSP3A, or RXSP3B/TXSP3B.
16	SP5I	**SPORT 5 Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the SP5I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). An SP5I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP5A/TXSP5A, RXSP5B/TXSP5B.
19–17	Reserved	
20	CB7I	**DAG1 Circular Buffer 7 Overflow Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the CB7I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). A circular buffer overflow occurs when the DAG circular buffering operation increments the I register past the end of the buffer. For more information, see "CB7S" on page A-20.
21	CB15I	**DAG2 Circular Buffer 15 Overflow Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the CB15I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). A circular buffer overflow occurs when the DAG circular buffering operation increments the I register past the end of the buffer. For more information, see "CB15S" on page A-20.
22	TMZLI	**Timer Expired (Low Priority) Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the TMZLI interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). For more information, see "TMZHI" on page A-27.
23	FIXI	**Fixed-Point Overflow Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the FIXI interrupt is unmasked (if set, = 1) or the FIXI interrupt is masked (if cleared, = 0). For more information, see "AOS" on page A-19.
24	FLTOI	**Floating-Point Overflow Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the FLTOI interrupt is unmasked (if set, = 1) or masked (if cleared, = 0).

Table A-11. IMASKP Register Bit Descriptions (Cont'd)

Bits	Name	Definition
25	FLTUI	**Floating-Point Underflow Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the FLTUI interrupt is unmasked (if set, = 1) or masked (if cleared, = 0).
26	FLTII	**Floating-Point Invalid Operation Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the FLTII interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). For more information, see "AIS" on page A-19.
27	EMULI	**Emulator (Lower Priority) Interrupt.** When the DSP is servicing another interrupt, this bit indicates if the EMULI interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). An EMULI occurs on reset and when an external device asserts the $\overline{\text{EMU}}$ pin. This interrupt has a lower priority than EMUI, but higher priority than software interrupts.
28	SFT0I	**User Software Interrupt 0.** When the DSP is servicing another interrupt, this bit indicates if the SFT0I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0). An SFT0I occurs when a program sets (= 1) this bit.
29	SFT1I	**User Software Interrupt 1.** When the DSP is servicing another interrupt, this bit indicates if the SFT1I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0).
30	SFT2I	**User Software Interrupt 2.** When the DSP is servicing another interrupt, this bit indicates if the SFT2I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0).
31	SFT3I	**User Software Interrupt 3.** When the DSP is servicing another interrupt, this bit indicates if the SFT3I interrupt is unmasked (if set, = 1) or masked (if cleared, = 0).

Interrupt Register (LIRPTL)

The LIRPTL register is a non-memory-mapped, universal, system register (*Ureg* and *Sreg*). The reset value for these registers is 0x0000 0000. The LIRPTL register indicates latch status, select masking, and displays mask pointers for interrupts. Figure A-14 and Table A-12 provide bit definitions for the LIRPTL register.

The MSKP bits in the LIRPTL register, and the entire IMASKP register are for interrupt controller use only. Modifying these bits interferes with the proper operation of the interrupt controller.

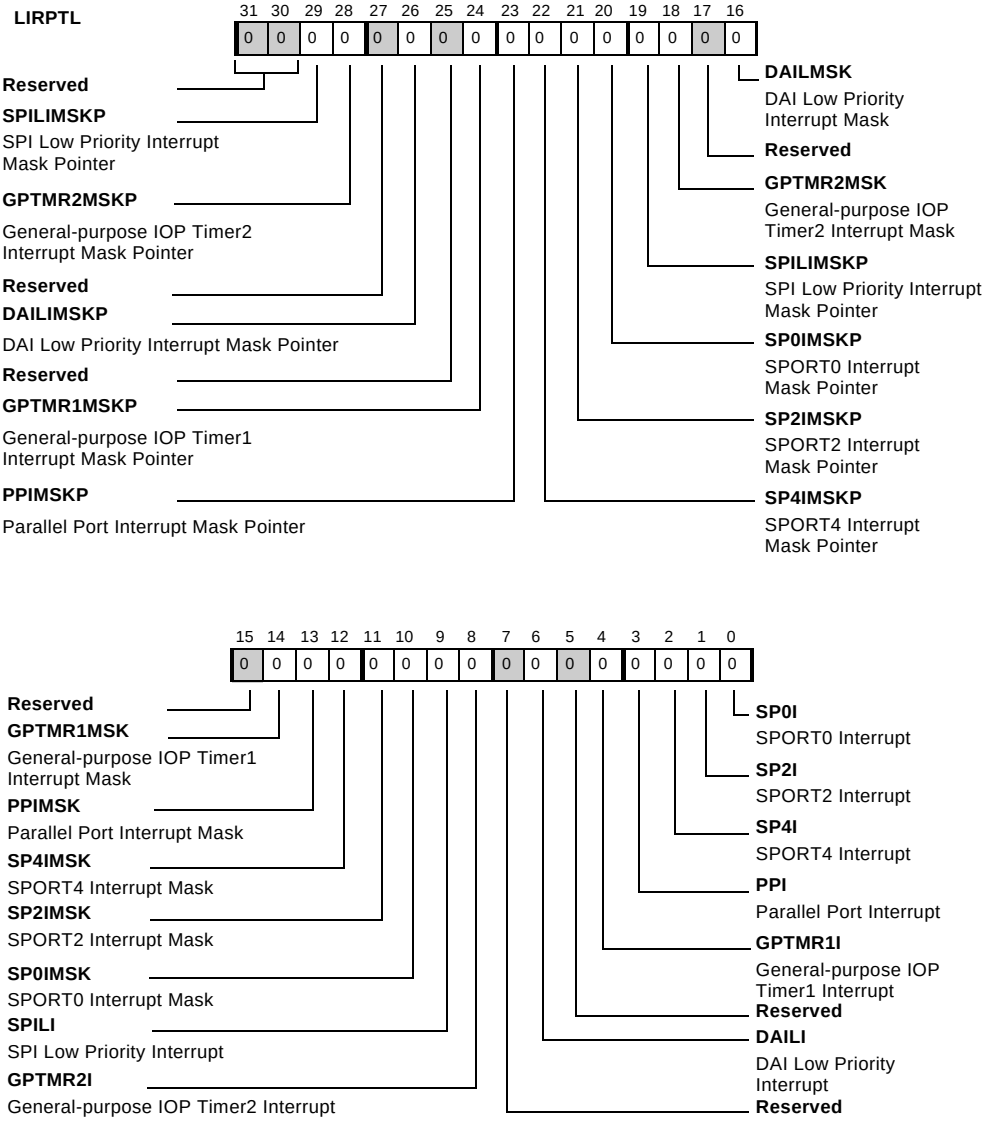


Figure A-14. LIRPTL Register

Table A-12. LIRPTL Register Bit Descriptions

Bits	Name	Definition
0	SP0I	**SPORT 0 Interrupt.** Indicates if an SP0I interrupt is latched and is pending (if set, = 1), or no SP0I is pending (if cleared, = 0). An SP0I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP0A/TXSP0A, RXSP0B/TXSP0B.
1	SP2I	**SPORT 2 Interrupt.** Indicates if an SP2I interrupt is latched and is pending (if set, = 1), or no SP2I is pending (if cleared, = 0). An SP2I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP2A/TXSP2A, RXSP2B/TXSP2B.
2	SP4I	**SPORT 4 Interrupt.** Indicates if an SP4I interrupt is latched and is pending (if set, = 1), or no SP4I is pending (if cleared, = 0). An SP4I interrupt occurs two cycles after the last bit of an input/output serial word is latched into/from RXSP4A/TXSP4A, RXSP4B/TXSP4B.
3	PPI	**Parallel Port Interrupt.** Indicates if a PP interrupt is latched and pending (if set, = 1) or that no PP interrupt is pending (if cleared, = 0). A PP interrupt occurs when the DMA block transfer has completed.
4	GPTMR1I	**General-Purpose IOP Timer 1 Interrupt.** Indicates if a GPTMR1 is latched and is pending (if set, = 1). If no GPTMR1I is pending (if cleared, = 0).
5	Reserved	
6	DAILI	**DAI Low Priority Interrupt.** Indicates if a DAI interrupt is latched and is pending (if set, = 1) or no DAI interrupt is pending (if cleared, = 0). This is the lower priority option.
7	Reserved	
8	GPTMR2I	**General-Purpose IOP Timer 2 Interrupt.** Indicates if a GPTMR2I is latched and is pending (if set, = 1) or no GPTMR2I is pending (if cleared, = 0).
9	SPILI	**SPI Interrupt (low priority).** Indicates if an SPIL interrupt is latched and pending (if set, = 1) or no SPIL interrupt is pending (if cleared, = 0).

Table A-12. LIRPTL Register Bit Descriptions (Cont'd)

Bits	Name	Definition
10	SP0IMSK	**SPORT0 Interrupt Mask.** Unmasks the SP0 interrupt (if set, = 1) or masks the SP0 interrupt (if cleared, = 0).
11	SP2IMSK	**SPORT2 Interrupt Mask.** Unmasks the SP2 interrupt (if set, = 1) or masks the SP2 interrupt (if cleared, = 0).
12	SP4IMSK	**SPORT4 Interrupt Mask.** Unmasks the SP4 interrupt (if set, = 1) or masks the SP4 interrupt (if cleared, = 0).
13	PPIMSK	**Parallel Port Interrupt Mask.** Unmasks the PP interrupt (if set, = 1) or masks the PP interrupt (if cleared, = 0).
14	GPTMR1IMSK	**General-Purpose IOP Timer 1 Interrupt Mask.** Unmasks the GPTMR1 interrupt (if set, = 1) or masks the GPTMR1 interrupt (if cleared, = 0).
15	Reserved	
16	DAILIMSK	**DAI Low Priority Interrupt Mask.** Unmasks the DAILI (if set, = 1) or masks DAILI (if cleared, = 0).
17	Reserved	
18	GPTMR2IMSK	**General-Purpose IOP Timer 2 Interrupt Mask.** Unmasks the GPTMR2 interrupt (if set, = 1) or masks the GPTMR2 interrupt (if cleared, = 0).
19	SPILIMSK	**SPI Interrupt Mask (Low Priority).** Unmasks the SPIL interrupt (if set, = 1) or masks the SPIL interrupt (if cleared, = 0). For more information on how interrupt masking works, see "Interrupt Mask Register (IMASK)" on page A-30.
20	SP0IMSKP	**SPORT0 Interrupt Mask Pointer.** When the DSP is servicing another interrupt, this bit indicates if the SP0 interrupt is unmasked (if set, = 1) or the SP0 interrupt is masked (if cleared, = 0).
21	SP2IMSKP	**SPORT2 Interrupt Mask Pointer.** When the DSP is servicing another interrupt, this bit indicates if the SP2 interrupt is unmasked (if set, = 1) or the SP2 interrupt is masked (if cleared, = 0).

Table A-12. LIRPTL Register Bit Descriptions (Cont'd)

Bits	Name	Definition
22	SP4IMSKP	**SPORT4 Interrupt Mask Pointer.** When the DSP is servicing another interrupt, this bit indicates if the SP4 interrupt is unmasked (if set, = 1) or the SP4 interrupt is masked (if cleared, = 0).
23	PPIMSKP	**Parallel Port Interrupt Mask Pointer.** When the DSP is servicing another interrupt, this bit indicates if the PP interrupt is unmasked (if set, = 1) or the PP interrupt is masked (if cleared, = 0).
24	GPTMR1MSKP	**General-Purpose IOP Timer 1 Interrupt Mask Pointer.** When the DSP is servicing another interrupt, this bit indicates if the GPTMR1 interrupt is unmasked (if set, = 1) or the GPTMR1 interrupt is masked (if cleared, = 0).
25	Reserved	
26	DAILIMSKP	**DAI Low Priority Interrupt Mask Pointer.** When the DSP is servicing another interrupt, this bit indicates if the DAILI is unmasked (if set, = 1) or masked (if cleared, = 0).
27	Reserved	
28	GPTMR2MSKP	**General-Purpose IOP Timer 2 Interrupt Mask Pointer.** When the DSP is servicing another interrupt, this bit indicates if the GPTMR2 interrupt is unmasked (if set, = 1) or the GPTMR2 interrupt is masked (if cleared, = 0). For more information on how interrupt mask pointers works, see "Interrupt Mask Register (IMASK)" on page A-30.
29	SPILIMSKP	**SPI Interrupt Mask (Low Priority) Pointer.** When the DSP is servicing another interrupt, this bit indicates if the SPIL interrupt is unmasked (if set, = 1) or the SPIL interrupt is masked (if cleared, = 0).For more information on how interrupt mask pointers works, see "Interrupt Mask Pointer Register (IMASKP)" on page A-35.
31–30	Reserved	

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 DSP 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 A-15, the address buses can handle 32-bit addresses, but the program sequencer only generates 24-bit addresses over the PM bus.

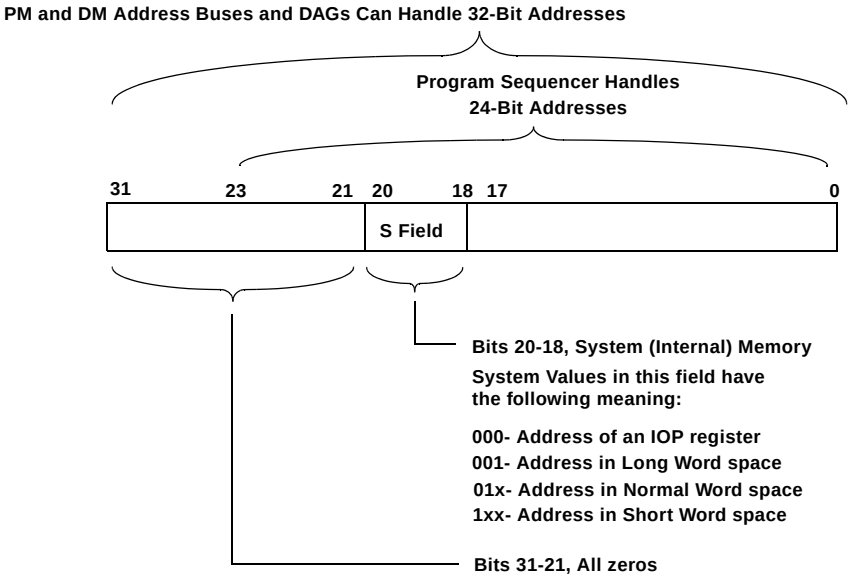


Figure A-15. PM and DM Bus Addresses Versus Sequencing Addresses

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 register is a readable and writable register.

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 DSP 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 DSP decodes on the next cycle.

Loop Address Stack Register (LADDR)

The LADDR register 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 A-13. LADDR Register Bit Descriptions

Bits	Value
23–0	Loop Termination Address
28–24	Termination Code
29	Reserved (always reads zero)
31–30	**Loop Type Code** 00 = arithmetic condition-based (not LCE) 01 = counter-based, length 1 10 = counter-based, length 2 11 = counter-based, length > 2

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 Counter Stack" on page 3-32.

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 Counter Stack" on page 3-32.

Timer Period Register (TPERIOD)

The TPERIOD register is a non memory-mapped, universal register (*Ureg* only). The Timer Period register contains the decrementing timer count value, counting down the cycles between timer interrupts. For more information on how to use the TPERIOD register, see "Timer and Sequencing" on page 3-44.

Timer Count Register (TCOUNT)

The TCOUNT register is a non memory-mapped, universal register (*Ureg* only). The Timer Count register contains the timer period, indicating the number of cycles between timer interrupts. For more information on how to use the TCOUNT register, see "Timer and Sequencing" on page 3-44.

Data Address Generator Registers

The DSP'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).

Table A-14. DAG Registers

Register	Initialization After Reset
"Index Registers (Ix)" on page A-51	Undefined
"Modify Registers (Mx)" on page A-51	Undefined
"Length and Base Registers (Lx,Bx)" on page A-51	Undefined

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" on page 4-1.

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" on page 4-1.

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" on page 4-1.

I/O Processor Registers

The I/O processor's registers are accessible as part of the DSP's memory map. These registers occupy addresses 0x0000 0000 through 0x0003 F FFF 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-2126x SHARC DSP is located in that model's peripherals manual. The I/O processor's memory-mapped registers are described in the *ADSP-21262/21266 SHARC DSP Peripherals Manual*.

Revision ID Register (REVPID)

The REVPID register is top layer metal programmable 8-bit register. Because REVPID register bits 7-0 are the DSP 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 DSP's processor ID and silicon revision numbers.

As shown in Table A-15, the bit position from 0–3 signifies the Processor-id. For ADSP-21262 processor, the process-id is 0000. The bit position 4–7 signifies the silicon revision-id. For the ADSP-21262 processor the present silicon revision -id is 0000.

Table A-15. REVPID Register Bit Descriptions

Bits	Name	Definition
3–0	PID	Processor Identification (Read-only) PID
7–4	Silicon Revision	Silicon Revision

Hardware Breakpoint Control Register (BRKCTL)

The BRKCTL register controls how breakpoints are used (if the UMODE bit is set). This user accessible register in the BRKCTL 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. The bits related to breakpoint register are same as in EMUCTL register.

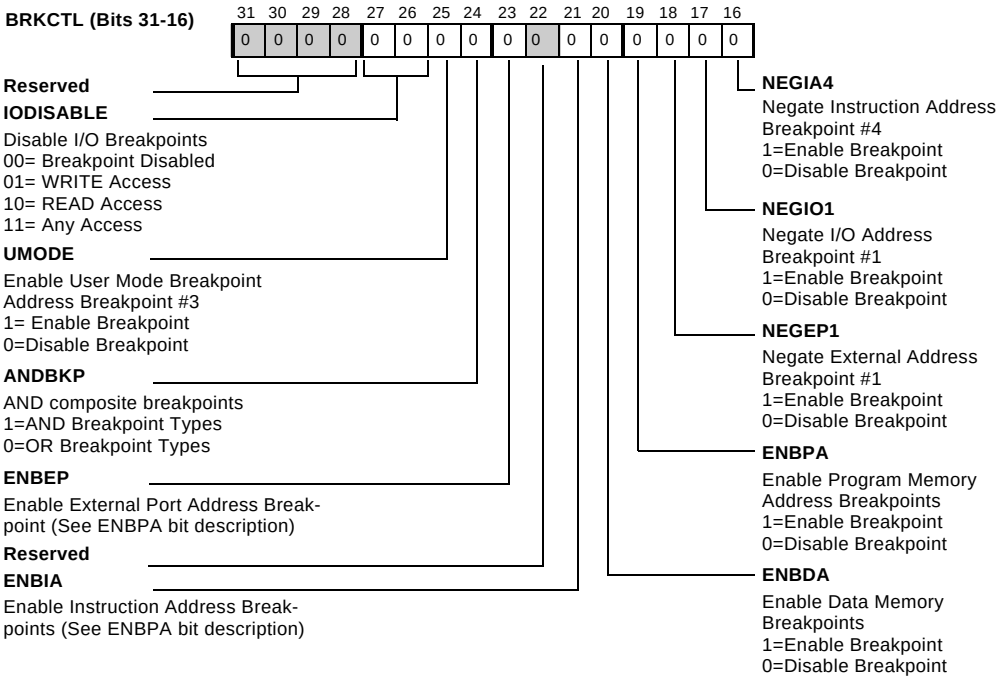


Figure A-16. BRKCTL Register (Upper Bits)

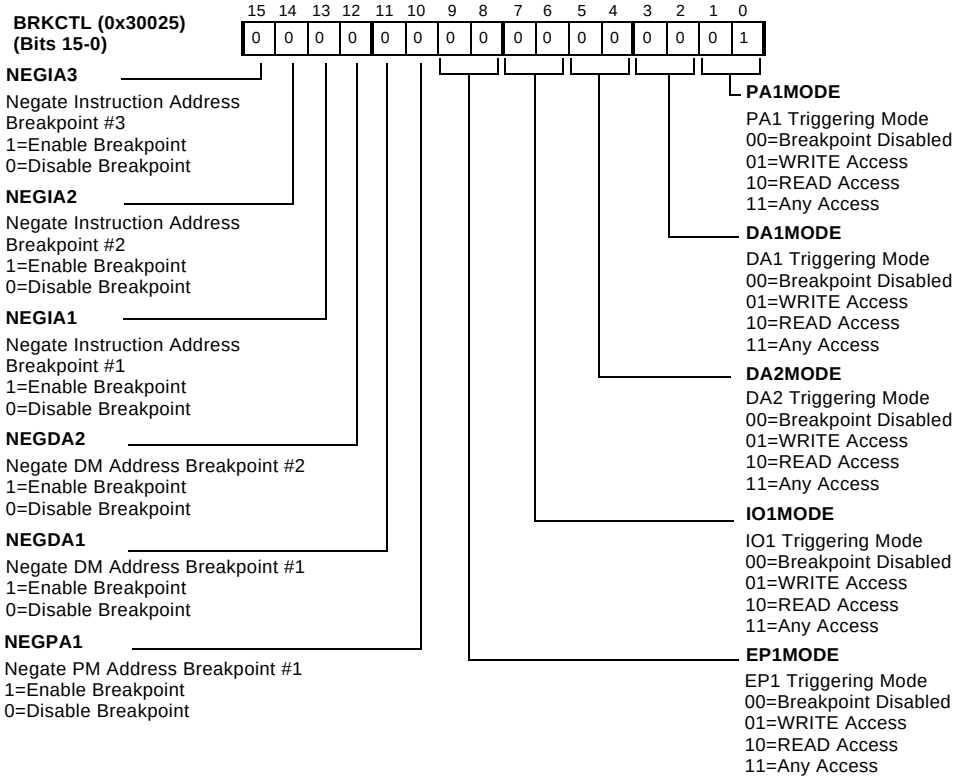


Figure A-17. BRKCTL Register (Lower Bits)

Table A-16. BRKCTL Register Bit Descriptions

Bit #	Name	Function
1–0	PA1MODE	**PA1Triggering Mode** 00 = Breakpoint Disabled 01 = WRITE Access 10 = READ Access 11 = Any access
3–2	DA1MODE	**DA1 Triggering Mode** 00 = Breakpoint Disabled 01 = WRITE Access 10 = READ Access 11 = Any access
5–4	DA2MODE	**DA2 Triggering Mode** 00 = Breakpoint Disabled 01 = WRITE Access 10 = READ Access 11 = Any Access
7–6	IO1MODE	**IO1 Triggering Mode** trigger on the following conditions: Mode Triggering condition 00 = Breakpoint is disabled 01 = WRITE accesses only 10 = READ accesses only 11 = Any access
9–8	EP1MODE	**EP1 Triggering Mode** 00 = Breakpoint Disabled 01 = WRITE Access 10 = READ Access 11 = Any Access
10	NEGPA1	**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
11	NEGDA1	**Negate Data Memory Address Breakpoint #1** For more information, see NEGPA1 bit description.

Table A-16. BRKCTL Register Bit Descriptions (Cont'd)

Bit #	Name	Function
12	NEGDA2	**Negate Data Memory Address Breakpoint #2** For more information, see NEGPA1 bit description.
13	NEGIA1	**Negate Instruction Address Breakpoint #1** 0 = Disable Breakpoint 1 = Enable Breakpoint
14	NEGIA2	**Negate Instruction Address Breakpoint #2** For more information, see NEGPA1 bit description.
15	NEGIA3	**Negate Instruction Address Breakpoint #3** For more information, see NEGPA1 bit description.
16	NEGIA4	**Negate Instruction Address Breakpoint #4** For more information, see NEGPA1 bit description.
17	NEGIO1	**Negate I/O Address Breakpoint** For more information, see NEGPA1 bit description.
18	NEGEP1	**Negate EP Address Breakpoint** For more information, see NEGPA1 bit description.
19	ENBPA	**Enable Program Memory Data Address Breakpoints** The ENB* bits 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
20	ENBDA	**Enable Data Memory Address Breakpoints** For more information, see ENBPA bit description.
21	ENBIA	**Enable Instruction Address Breakpoints.** For more information, see ENBPA bit description.
22	Reserved	
23	ENBEP	**Enable External Port Address Breakpoint.** For more information, see ENBPA bit description.

Table A-16. BRKCTL Register Bit Descriptions (Cont'd)

Bit #	Name	Function
24	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
25	UMODE	**User Mode Breakpoint Functionality Enable** Address Breakpoint 3 0 = Disable Breakpoint 1 = Enable Breakpoint
27–26	IODISABLE	**Enable I/O Breakpoints** 00 = Breakpoint Disabled 01 = WRITE Access 10 = READ Access 11 = Any Access
31–28	Reserved	

Enhanced Emulation Status Register (EEMUSTAT)

The `EEMUSTAT` register reports the breakpoint status of the programs that run on the ADSP-21262 processor. This register is a memory-mapped IOP register that can be accessed by the core. This register contains two status bits that report I/O breakpoints, one each for the two I/O buses (`IOX` and `IOY`).

When a breakpoint is reached, an interrupt occurs and the breakpoint's status bits are set. When the core returns from an interrupt, the breakpoint's status bits are cleared. Figure A-18 lists this register's bits.

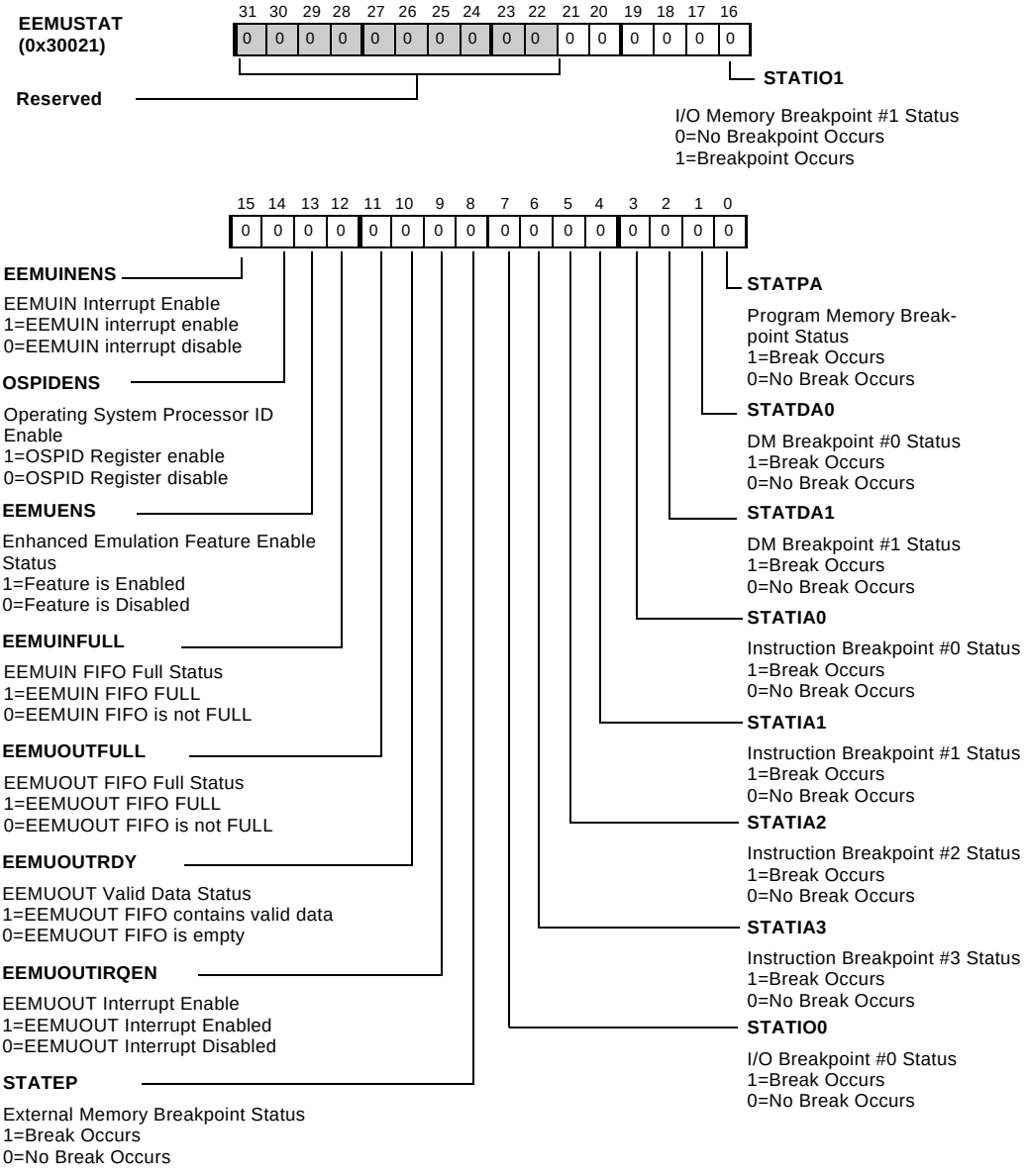


Figure A-18. EEMUSTAT Register

Table A-17. EEMUSTAT Register Definitions

Bits	Name	Function
0	STATPA	**Program memory Data Breakpoint Hit.**[1] 1= Program memory breakpoint occurs 0= No program memory breakpoint occurs
1	STATDA0	**Data Memory Breakpoint Hit.**[1] 1= Data memory #0 breakpoint occurs 0= No Data memory #0 breakpoint occurs
2	STATDA1	**Data Memory Breakpoint Hit**.[1] 1= Data memory #1 breakpoint occurs 0= No Data memory #1 breakpoint occurs
3	STATIA0	**Instruction Address Breakpoint Hit.**[1] 1= Instruction address #0 breakpoint occurs 0= no Instruction address #0 breakpoint occurs
4	STATIA1	**Instruction Address Breakpoint Hit.**[1] 1= Instruction address #1 breakpoint occurs 0= no Instruction address #1 breakpoint occurs
5	STATIA2	**Instruction Address Breakpoint Hit.**[1] 1= Instruction address #2 breakpoint occurs 0= no Instruction address #2 breakpoint occurs
6	STATIA3	**Instruction Address Breakpoint Hit.**[1] 1 = Instruction address #3 breakpoint occurs 0= no Instruction address #3 breakpoint occurs
7	STATIO	**I/O Address Breakpoint Hit.**[1] 1= I/OX address breakpoint occurs 0= no I/OX address breakpoint occurs
8	Reserved[1]	
9	EEMUOUTIRQEN	**Enhanced Emulation EEMUOUT Interrupt Enable.**[2] 1 = EEMUOUT interrupt enable 0 = EEMUOUT interrupt disable Note: Interrupts are of low priority interrupts

Table A-17. EEMUSTAT Register Definitions (Cont'd)

Bits	Name	Function
10	EEMUOUTRDY	**Enhanced Emulation EEMUOUT Ready.**[3] 1= EEMUOUT FIFO contains valid data 0= EEMUOUT FIFO is empty
11	EEMUOUTFULL	**Enhanced Emulation EEMUOUT FIFO Status.**[3] 1= EEMUOUT FIFO FULL 0= EEMUOUT FIFO is not FULL
12	EEMUINFULL	**Enhanced Emulation EEMUIN Register Status.**[4] 1= EEMUIN register full 0= EEMUIN register is empty
13	EEMUENS	**Enhanced Emulation Feature Enable.**[4] 1= Enhanced emulation feature enable 0= Enhanced emulation feature disable
14	OSPIDENS	**OSPID Register Enable.**[4] 1= OSPID register enable 0= OSPID register disable
15	EEMUINENS	**EEMUIN Interrupt Enable.**[4] 1= EEMUIN interrupt enable 0= EEMUIN interrupt disable
16	STATIO1	**I/O Memory Breakpoint 1 Status** 0= No Breakpoint Occurs 1= Breakpoint Occurs
31:17	Reserved for future use.	

1 Internal hardware sets this bit.
2 This bit is set and reset by the core.
3 The FIFO controller sets and resets this bit.
4 Internal hardware sets and resets this bit.

Timer Registers

The ADSP-21262 processor Timer peripheral module provides general-purpose timer functionality. It consists of three identical Timer units.

To provide the required functionality, each Timer has four 32-bit memory-mapped registers. The registers for each timer are:

- Timer x Configuration (TMxCTL) registers, described on page A-61
- Timer x Word Count (TMxCNT) registers, described on page A-62
- Timer x Word Period (TMxPRD) registers, described on page A-63
- Timer x Word Pulsewidth (TMxW) registers, described on page A-63

The timers also share one common status and control register:

- Timer Global Status and Control (TMSTAT) register, described on page A-64

Timer Configuration Registers (TMxCTL)

The three TMxCTL registers' addresses are: TM0CTL 0x1401, TM1CTL 0x1409, TM2CTL 0x1411. All Timer clocks are gated OFF when the specific Timer's configuration register is set to zero at system reset or subsequently reset by the user.

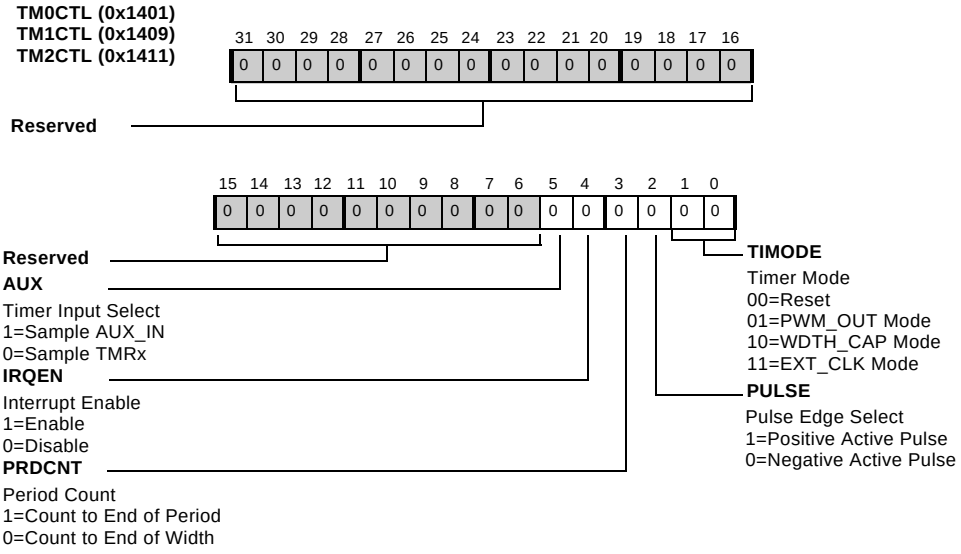


Figure A-19. Timer Configuration Register

Timer Counter Registers (TMxCNT)

The TMxCNT registers addresses are: TM0CNT 0x1402, TM1CNT 0x140A, TM2CNT 0x1412. When disabled, the Timer counter retains its state. When enabled again, the Timer counter is re-initialized 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 TMxPRD registers' addresses are: TM0PRD 0x1403, TM1PRD 0x140B, TM2PRD 0x1413. Once a timer is enabled and running, when the DSP 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 TMxW registers' addresses 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 this address: 0x1400. Status bits are sticky and require a write-one to clear operation. During a status register read access, all reserved or unused bits will return a zero. The reset state is 0x0000. Each Timer generates a unique DSP 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 re-issued.

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.

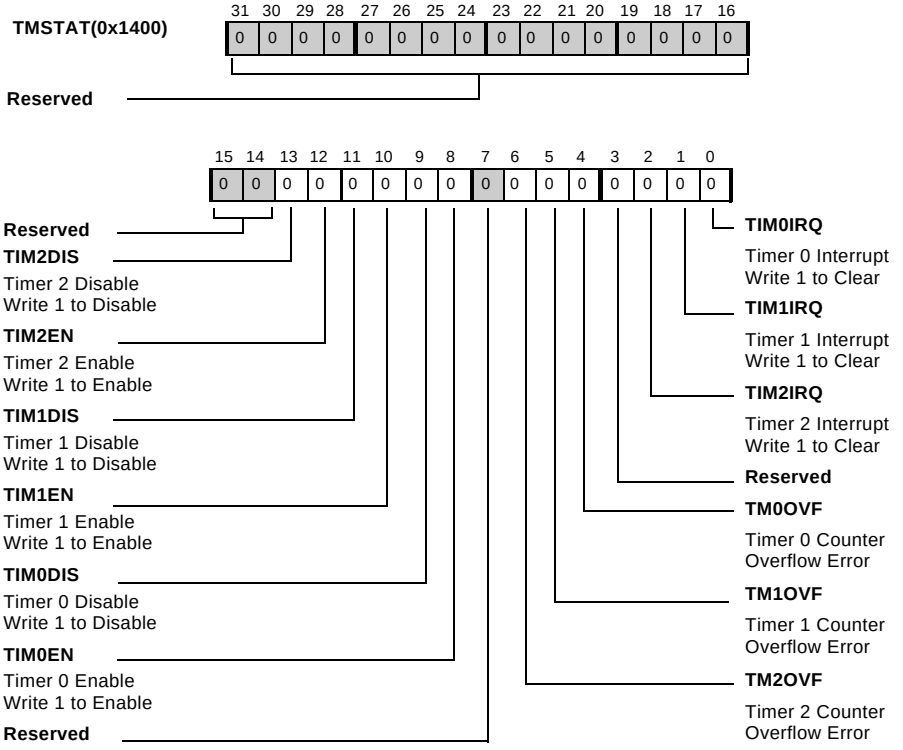


Figure A-20. TMSTAT Register

Power Management Registers

The following sections describe the registers associated with the DSPs power management functions.

Table A-18. Timer Global Status and Control (TMSTAT) Register Bits

Bit(s)	Name	Definition
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 will return a zero.

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 phase lock loop (PLL) multiplier and Divider (both input and output) values, PLL bypass mode, and clock enabling control for peripherals (see Table A-19). 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.

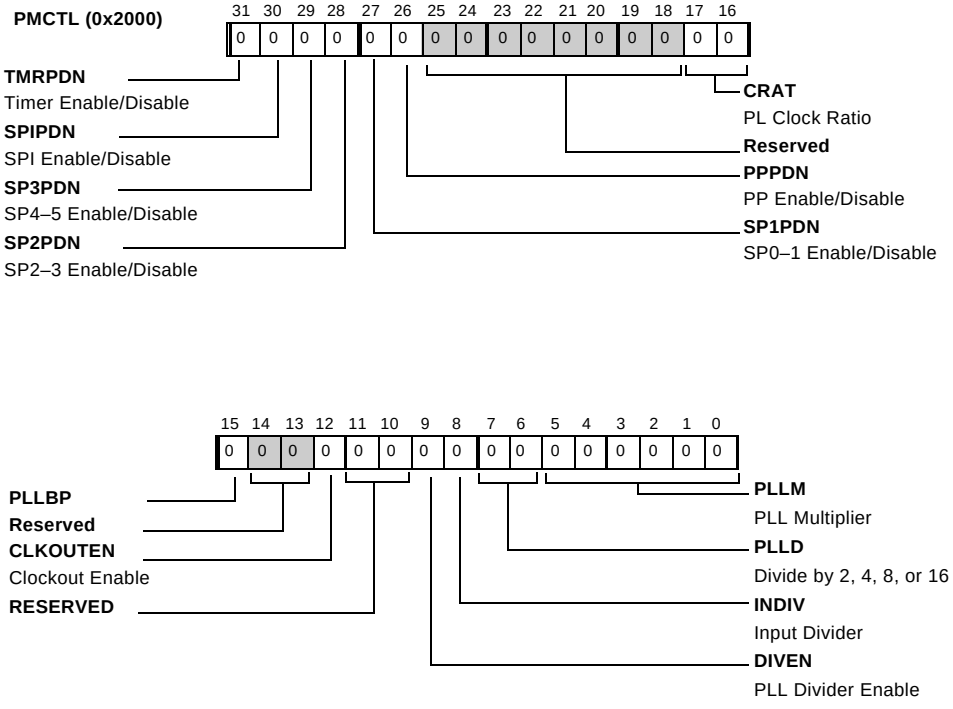


Figure A-21. PMCTL Register

Table A-19. PMCTL Register Bit Descriptions

Bits	Name	Definition
5:0	PLLM	**PLL Multiplier.** Read/Write PLLM = 0 PLL Multiplier = 64 0<PLLM<63 PLL Multiplier = PLLM CLK_CFG[1:0] Reset Value 00 = 0000110 01 = 100000 10 = 010000 11 = 000110
7:6	PLLDx	**PLL Divider.** Read/Write 00 = CK divider = 2 01 = CK divider = 4 10 = CK divider = 8 11 = CK divider = 16 CLK_CFG[1:0] 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 A-19. PMCTL Register Bit Descriptions (Cont'd)

Bits	Name	Definition
17:16	CRAT	**PLL clock ratio (CLKIN to CK).** Read only. For more detail look for refer to the ADSP-21262 processor clock configuration pin description. Reset Value = CLK_CFG[1:0]
25–18	Reserved	
26	PPPDN	**PP Enable/Disable.** Read/Write 0 = PP is in normal mode 1 = Shutdown clock to PP 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
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 FLGx pins cannot be used (via the FLGS7–0 register bits) because the FLGx pins are synchronized with the clock. Reset Value = 0
31	TMRPDN	**Timer Enable/Disable.** Read/Write 0 = Timer is in normal mode 1 = Shutdown clock to Timer Reset Value = 0

B INTERRUPT VECTOR ADDRESSES

Table B-2 shows all the ADSP-2126x DSP interrupts, listed according to their bit position in the IRPTL, LIRPTL, and IMASK registers. For more information, see "Interrupt Latch Register (IRPTL)" on page A-25, "Interrupt Register (LIRPTL)" on page A-42, and "Interrupt Mask Register (IMASK)" on page A-30. Also shown is the address of the interrupt vector. Each vector is separated by four memory locations. The addresses in the vector table represent offsets from a base address. For an Interrupt Vector Table in internal RAM, the base address is 0x8 0000 and for internal ROM, the base address is 0xA 0000. These are 48-bit addresses.

Table B-1. Interrupt Vector Table Base Address

Address[1]	Description
0x0008 0000	Internal RAM
0x000A 0000	Internal ROM

1 These are 48-bit addresses.

The interrupt name column in Table B-2 lists a mnemonic name for each interrupt as they are defined by the definitions file (def2126x.h) that comes with the software development tools.

SPI has only one interrupt for both transmit and receive.

Each serial port (SPORT) has only one interrupt for both transmit and receive.

Table B-2. ADSP-2126x Interrupt Vector Addresses

Register	IRPTL/ IMASK, LIRPTL Bit#	Vector Address	Interrupt Name	Function
IRPTL	0	0x00	EMUI	Emulator (read-only, non-maskable); HIGHEST PRIORITY
IRPTL	1	0x04	RSTI	Reset (read-only, non-maskable)
IRPTL	2	0x08	IICDI	Illegal Input Condition Detected
IRPTL	3	0x0C	SOVFI	Status loop or mode stack overflow; or PC stack full
IRPTL	4	0x10	TMZHI	Timer = 0 (high priority option)
IRPTL	5	0x14		Reserved
IRPTL	6	0x18	BKPI	Hardware Breakpoint Interrupt
IRPTL	7	0x1C		Reserved
IRPTL	8	0x20	IRQ2I	IRQ2I_ is asserted
IRPTL	9	0x24	IRQ1I	IRQ1I_ is asserted
IRPTL	10	0x28	IRQ0I	IRQ0I_ is asserted
IRPTL	11	0x2C	DAIHI	DAI High Priority Interrupt
IRPTL	12	0x30	SPIHI	SPI Transmit or Receive (higher priority option)
IRPTL	13	0x34	GPTMR0I	General-purpose IOP Timer 0 Interrupt
IRPTL	14	0x38	SP1I	SPORT1 Interrupt
IRPTL	15	0x3C	SP3I	SPORT3 Interrupt
IRPTL	16	0x40	SP5I	SPORT5 Interrupt
LIRPTL	0	0x44	SP0I	SPORT0 Interrupt
LIRPTL	1	0x48	SP2I	SPORT2 Interrupt
LIRPTL	2	0x4C	SP4I	SPORT4 Interrupt

Table B-2. ADSP-2126x Interrupt Vector Addresses (Cont'd)

Register	IRPTL/ IMASK, LIRPTL Bit#	Vector Address	Interrupt Name	Function
LIRPTL	3	0x50	PPI	Parallel Port Interrupt
LIRPTL	4	0x54	GPTMR1I	General-purpose IOP Timer 1 Interrupt
LIRPTL	5	0x58	--	Reserved
LIRPTL	6	0x5C	DAILI	DAI Low Priority Interrupt
LIRPTL	7	0x60	--	Reserved
IRPTL	17, 18, 19	0x64-0x6F	--	Reserved
LIRPTL	8	0x70	GPTMR2I	General-purpose IOP Timer 2 Interrupt
LIRPTL	9	0X74	SPILI	SPI Transmit or Receive (lower priority option)
IRPTL	20	0x78	CB7I	Circular Buffer 7 Overflow
IRPTL	21	0x7C	CB15I	Circular Buffer 15 Overflow
IRPTL	22	0x80	TMZLI	Timer=0(Low Priority Option)
IRPTL	23	0x84	FIXI	Fixed-point Overflow
IRPTL	24	0x88	FLTOI	Floating-point Overflow Exception
IRPTL	25	0x8C	FLTUI	Floating-point Underflow Exception
IRPTL	26	0x90	FLTII	Floating-point invalid exception
IRPTL	27	0x94	EMULI	Emulator Low Priority Interrupt
IRPTL	28	0x98	SFT0I	User Software Interrupt 0
IRPTL	29	0x9C	SFT1I	User Software Interrupt 1
IRPTL	30	0xA0	SFT2I	User Software Interrupt 2
IRPTL	31	0xA4	SFT3I	User Software Interrupt 3; LOWEST PRIORITY

G GLOSSARY

Autobuffering Unit (ABU). See I/O processor on page G-5 and DMA on page G-3.

Arithmetic Logic Unit (ALU). This part of a processing element performs arithmetic and logic operations on fixed-point and floating-point data.

Auxiliary registers. See Index Registers on page G-5.

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 registers. 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 Do/Until instructions in the *ADSP-21160 DSP Instruction Set Reference.*

Block size register. See Length Registers on page G-6.

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-9.

Bus slave or slave mode. A DSP can be a bus slave to another DSP or to a host processor. The DSP becomes a host bus slave when the $\overline{\text{HBG}}$ signal is returned.

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.

DAGEN, Data address generator. See Data Address Generator (DAG).

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 DSP 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 DSP.

Delayed branches. These are JUMPS and CALL/return instructions with the delayed branches (DB) modifier. In delayed branches, no instruction cycles

are lost in the pipeline, because the DSP 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`/`return` instructions that use an absolute—not changing at runtime—address (such as a program label) or use a PC-relative address.

Direct reads & writes. A direct access of the DSP's internal memory or I/O processor registers by another DSP or by a host processor.

DMA (Direct Memory Accessing). The DSP's I/O processor supports DMA of data between DSP 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 DSP 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 DSP 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 DSP 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 DSP supports both big- and little-endian bit order data transfers. Also for compatibility little endian hosts, the DSP supports both big and little endian word order data transfers.

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).

Programmable Flag pins. 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.

General purpose input/output pins. See Programmable Flag pins.

Flag update. The DSP'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.

Harvard architecture. DSPs use memory architectures that have separate buses for program and data storage. The two buses let the DSP get a data word and an instruction simultaneously.

Hold time cycle. This is an inactive bus cycle that the DSP 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 DSP's on-chip I/O processor.

Idle cycle. This is an inactive bus cycle that the DSP 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`/return instructions that use a dynamic—changes at runtime—address that comes from the PM data address generator.

Inexact flags. An inexact flag is an exception flag whose bit position is inexact.

Interleaved data. To take advantage of the DSP'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 DSP'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.

Length registers. A length registers is a Data Address Generator (DAG) register that sets up the range of addresses a circular buffer.

Level-sensitive interrupts. The DSP 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 port.

MCM, Multichannel mode. See Multichannel mode on page G-9.

Memory Access Modes. The DSP supports Asynchronous external memory space. In asynchronous access mode, the DSP's $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes change before CLKIN edge. In synchronous access mode, the DSP's $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes change on CLKIN edge.

Memory blocks and banks. The DSP's internal memory is divided into **blocks** that are each associated with different data address generators. The DSP'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 DSP 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 DSP 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 DSP 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 DSPs 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-21535's peripherals include internal memory, parallel port, I/O processor, JTAG port, and any external devices that connect to the DSP.

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 DSP 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.

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.

Pre-modify addressing. The Data Address Generator (DAG) provides a modified address during a data move without incrementing the stored address.

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.

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 ports. The DSP 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 DSP 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.

SMUL, Saturation on multiplication. See Multiplier on page G-7.

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.

SST, Saturation on store. See Multiplier on page G-7.

Subroutines. The processor temporarily interrupts sequential flow to execute instructions from another part of program memory.

TADD, TDM address. See Multichannel Mode on page G-7.

TCB chain loading. The process in which the DSP'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 DSP's DMA controller for chained DMA operations.

Tristate Versus Three-state. 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-DSP) 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.

I INDEX

M

N

R

S

T

U

V

W

X

Z